ADSP-BF59x Blackfin® Processor Hardware Reference

Revision 1.0, May 2011

Part Number
82-100102-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

© 2011 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, EZ-KIT Lite, SHARC, TigerSHARC, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual xxxi
Intended Audience xxxi
Manual Contents xxxii
What's New in This Manual xxxiv
Technical or Customer Support xxxiv
Registration for MyAnalog.com xxxv
EngineerZone xxxv
Social Networking Web Sites xxxvi
Supported Processors xxxvi
Product Information xxxvii
Analog Devices Web Site xxxvii
VisualDSP++ Online Documentation xxxviii
Technical Library CD xxxviii
Notation Conventions xxxix

INTRODUCTION

General Description of Processor 1-1
Portable Low-Power Architecture 1-2

Peripherals 1-3
Memory Architecture 1-4
Internal Memory 1-5
I/O Memory Space 1-5
DMA Support 1-6
General-Purpose I/O (GPIO) 1-7
Two-Wire Interface 1-8
Parallel Peripheral Interface 1-9
SPORT Controllers 1-11
Serial Peripheral Interface (SPI) Ports 1-13
Timers 1-13
UART Port 1-14
Watchdog Timer 1-15
Clock Signals 1-16
Dynamic Power Management 1-16
Full-On Mode (Maximum Performance) 1-17
Active Mode (Moderate Power Savings) 1-17
Sleep Mode (High Power Savings) 1-17
Deep Sleep Mode (Maximum Power Savings) 1-18
Hibernate State 1-18
Instruction Set Description 1-18
Development Tools 1-19

MEMORY

Memory Architecture 2-1
L1 Instruction SRAM 2-2
L1 Instruction ROM 2-3
L1 Data SRAM 2-3
Boot ROM 2-4
External Memory 2-4
Processor-Specific MMRs 2-4
DTEST_COMMAND Register 2-5
ITEST_COMMAND Register 2-6
DMEM_CONTROL Register 2-7
IMEM_CONTROL Register 2-7
DCPLB_DATAx Registers 2-8
ICPLB_DATAx Registers 2-9

CHIP BUS HIERARCHY

Chip Bus Hierarchy Overview 3-1
Interface Overview 3-2
Internal Clocks 3-3
Core Bus Overview 3-3
Peripheral Access Bus (PAB) 3-4
PAB Arbitration 3-5
PAB Agents (Masters, Slaves) 3-5
PAB Performance 3-6

DMA Access Bus (DAB), DMA Core Bus (DCB) 3-6
DAB and DCB Arbitration 3-6
DAB Bus Agents (Masters) 3-7
DAB and DCB Performance 3-8

SYSTEM INTERRUPTS

Specific Information for the ADSP-BF59x 4-1
Overview 4-1
Features 4-2
Description of Operation 4-2
Events and Sequencing 4-2
System Peripheral Interrupts 4-4
Programming Model 4-7
System Interrupt Initialization 4-7
System Interrupt Processing Summary 4-8
System Interrupt Controller Registers 4-10
System Interrupt Assignment (SIC_IAR) Register 4-10
System Interrupt Mask (SIC_IMASK) Register 4-12
System Interrupt Status (SIC_ISR) Register 4-12
System Interrupt Wakeup-Enable (SIC_IWR) Register 4-12
Programming Examples 4-13
Clearing Interrupt Requests 4-13
Unique Information for the ADSP-BF59x Processor 4-15
Interfaces 4-15
System Peripheral Interrupts 4-17

DIRECT MEMORY ACCESS

Specific Information for the ADSP-BF59x 5-1
Overview and Features 5-2
DMA Controller Overview 5-4
External Interfaces 5-4
Internal Interfaces 5-4
Peripheral DMA 5-5
Memory DMA 5-6
Handshaked Memory DMA (HMDMA) Mode 5-8
Modes of Operation 5-9
Register-Based DMA Operation 5-9
Stop Mode 5-11
Autobuffer Mode 5-11
Two-Dimensional DMA Operation 5-11
Examples of Two-Dimensional DMA 5-12
Descriptor-based DMA Operation 5-13
Descriptor List Mode 5-14
Descriptor Array Mode 5-15
Variable Descriptor Size 5-15
Mixing Flow Modes 5-16
Functional Description 5-17
DMA Operation Flow 5-17
DMA Startup 5-17
DMA Refresh 5-22

Work Unit Transitions 5-24
DMA Transmit and MDMA Source 5-25
DMA Receive 5-26
Stopping DMA Transfers 5-28
DMA Errors (Aborts) 5-28
DMA Control Commands 5-31
Restrictions 5-34
Transmit Restart or Finish 5-34
Receive Restart or Finish 5-35
Handshaked Memory DMA Operation 5-36
Pipelining DMA Requests 5-37
HMDMA Interrupts 5-39
DMA Performance 5-40
DMA Throughput 5-41
Memory DMA Timing Details 5-44
Static Channel Prioritization 5-44
Temporary DMA Urgency 5-44
Memory DMA Priority and Scheduling 5-46
Traffic Control 5-48
Programming Model 5-50
Synchronization of Software and DMA 5-50
Single-Buffer DMA Transfers 5-52
Continuous Transfers Using Autobuffering 5-53
Descriptor Structures 5-55

Descriptor Queue Management 5-56
Descriptor Queue Using Interrupts on Every Descriptor 5-57
Descriptor Queue Using Minimal Interrupts 5-58
Software Triggered Descriptor Fetches 5-60
DMA Registers 5-62
DMA Channel Registers 5-62
DMA Peripheral Map Registers DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP) 5-66
DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG) 5-67
DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS) 5-71
DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR) .. 5-74
DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR) 5-74
DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT) 5-75
DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT) 5-76
DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY) 5-77
DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT) 5-78
DMA Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/ MDMA_yy_CURR_Y_COUNT) 5-78

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY) 5-79
DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR) 5-80
DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR) 5-81
HMDMA Registers 5-82
Handshake MDMA Control Registers (HMDMAx_CONTROL) 5-82
Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT) 5-84
Handshake MDMA Current Block Count Registers (HMDMAx_BCOUNT) 5-84
Handshake MDMA Current Edge Count Registers (HMDMAx_ECOUNT) 5-85
Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT) 5-86
Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT) 5-87
Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW) 5-87
DMA Traffic Control Registers (DMA_TC_PER and DMA_TC_CNT) 5-88
DMA_TC_PER Register 5-88
DMA_TC_CNT Register 5-88
Programming Examples 5-90

Register-Based 2-D Memory DMA 5-90
Initializing Descriptors in Memory 5-93
Software-Triggered Descriptor Fetch Example 5-96
Handshaked Memory DMA Example 5-99
Unique Information for the ADSP-BF59x Processor 5-101
Static Channel Prioritization 5-101

DYNAMIC POWER MANAGEMENT

Phase Locked Loop and Clock Control 6-1
PLL Overview 6-2
PLL Clock Multiplier Ratios 6-3
Core Clock/System Clock Ratio Control 6-5
Dynamic Power Management Controller 6-7
Operating Modes 6-7
Dynamic Power Management Controller States 6-8
Full-On Mode 6-8
Active Mode 6-8
Sleep Mode 6-9
Deep Sleep Mode 6-9
Hibernate State 6-10
Operating Mode Transitions 6-10
Programming Operating Mode Transitions 6-13
Dynamic Supply Voltage Control 6-15
Power Supply Management 6-15
Changing Voltage 6-15

Powering Down the Core (Hibernate State) 6-17
PLL and VR Registers 6-18
PLL_DIV Register 6-19
PLL_CTL Register 6-20
PLL_STAT Register 6-20
PLL_LOCKCNT Register 6-21
VR_CTL Register 6-21
System Control ROM Function 6-22
Programming Model 6-24
Accessing the System Control ROM Function in C/C++ 6-24
Accessing the System Control ROM Function in Assembly 6-25
Programming Examples 6-28
Full-on Mode to Active Mode and Back 6-30
Transition to Sleep Mode or Deep Sleep Mode 6-32
Set Wakeup Events and Enter Hibernate State 6-33
Perform a System Reset or Soft-Reset 6-35
In Full-on Mode, Change VCO Frequency, Core Clock Frequency, and System Clock Frequency 6-36
Changing Voltage Levels 6-38

GENERAL-PURPOSE PORTS

Overview 7-1
Features 7-1
Interface Overview 7-2
External Interface 7-3

Port F Structure 7-3
Port G Structure 7-4
Additional Considerations 7-5
Internal Interfaces 7-6
Performance/Throughput 7-6
Description of Operation 7-7
Operation 7-7
General-Purpose I/O Modules 7-8
GPIO Interrupt Processing 7-11
Programming Model 7-17
GPIO Schmitt Trigger Control 7-19
PORTx Pad Control Registers 7-19
Memory-Mapped GPIO Registers 7-20
Port Multiplexer Control Register (PORTx_MUX) 7-21
Function Enable Registers (PORTx_FER) 7-22
GPIO Direction Registers (PORTxIO_DIR) 7-22
GPIO Input Enable Registers (PORTxIO_INEN) 7-23
GPIO Data Registers (PORTxIO) 7-23
GPIO Set Registers (PORTxIO_SET) 7-24
GPIO Clear Registers (PORTxIO_CLEAR) 7-24
GPIO Toggle Registers (PORTxIO_TOGGLE) 7-25
GPIO Polarity Registers (PORTxIO_POLAR) 7-25
Interrupt Sensitivity Registers (PORTxIO_EDGE) 7-26
GPIO Set on Both Edges Registers (PORTxIO_BOTH) 7-26

GPIO Mask Interrupt Registers (PORTxIO_MASKA/B) 7-27
GPIO Mask Interrupt Set Registers (PORTxIO_MASKA/B_SET) 7-28
GPIO Mask Interrupt Clear Registers (PORTxIO_MASKA/B_CLEAR) 7-30
GPIO Mask Interrupt Toggle Registers (PORTxIO_MASKA/B_TOGGLE) 7-32
Programming Examples 7-33

GENERAL-PURPOSE TIMERS

Specific Information for the ADSP-BF59x 8-1
Overview 8-2
External Interface 8-3
Internal Interface 8-4
Description of Operation 8-4
Interrupt Processing 8-5
Illegal States 8-7
Modes of Operation 8-10
Pulse Width Modulation (PWM_OUT) Mode 8-10
Output Pad Disable 8-12
Single Pulse Generation 8-12
Pulse Width Modulation Waveform Generation 8-13
PULSE_HI Toggle Mode 8-15
Externally Clocked PWM_OUT 8-20
Using PWM_OUT Mode With the PPI 8-21
Stopping the Timer in PWM_OUT Mode 8-21

Pulse Width Count and Capture (WDTH_CAP) Mode 8-23
Autobaud Mode 8-31
External Event (EXT_CLK) Mode 8-32
Programming Model 8-33
Timer Registers 8-34
Timer Enable Register (TIMER_ENABLE) 8-35
Timer Disable Register (TIMER_DISABLE) 8-36
Timer Status Register (TIMER_STATUS) 8-38
Timer Configuration Register (TIMER_CONFIG) 8-40
Timer Counter Register (TIMER_COUNTER) 8-41
Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers 8-42
Summary 8-45
Programming Examples 8-47
Unique Information for the ADSP-BF59x Processor 8-56
Interface Overview 8-57
External Interface 8-58

CORE TIMER

Specific Information for the ADSP-BF59x 9-1
Overview and Features 9-1
Timer Overview 9-2
External Interfaces 9-2
Internal Interfaces 9-3
Description of Operation 9-3

Interrupt Processing 9-3
Core Timer Registers 9-4
Core Timer Control Register (TCNTL) 9-5
Core Timer Count Register (TCOUNT) 9-5
Core Timer Period Register (TPERIOD) 9-6
Core Timer Scale Register (TSCALE) 9-7
Programming Examples 9-7
Unique Information for the ADSP-BF59x Processor 9-9

WATCHDOG TIMER

Specific Information for the ADSP-BF59x 10-1
Overview and Features 10-1
Interface Overview 10-3
External Interface 10-3
Internal Interface 10-3
Description of Operation 10-4
Register Definitions 10-5
Watchdog Count (WDOG_CNT) Register 10-5
Watchdog Status (WDOG_STAT) Register 10-6
Watchdog Control (WDOG_CTL) Register 10-7
Programming Examples 10-8
Unique Information for the ADSP-BF59x Processor 10-10

UART PORT CONTROLLERS

Specific Information for the ADSP-BF59x 11-1

Overview 11-2
Features 11-2
Interface Overview 11-3
External Interface 11-3
Internal Interface 11-4
Description of Operation 11-5
UART Transfer Protocol 11-5
UART Transmit Operation 11-6
UART Receive Operation 11-7
IrDA Transmit Operation 11-8
IrDA Receive Operation 11-9
Interrupt Processing 11-11
Bit Rate Generation 11-12
Autobaud Detection 11-13
Programming Model 11-15
Non-DMA Mode 11-15
DMA Mode 11-17
Mixing Modes 11-18
UART Registers 11-19
UART Line Control (UART_LCR) Register 11-21
UART Modem Control (UART_MCR) Register 11-23
UART Line Status (UART_LSR) Register 11-24
UART Transmit Holding (UART_THR) Register 11-25
UART Receive Buffer (UART_RBR) Register 11-26

UART Interrupt Enable (UART_IER) Register 11-26
UART Interrupt Identification (UART_IIR) Register 11-28
UART Divisor Latch (UART_DLL and UART_DLH) Registers 11-29
UART Scratch (UART_SCR) Register 11-30
UART Global Control (UART_GCTL) Register 11-31
Programming Examples 11-32
Unique Information for the ADSP-BF59x Processor 11-41

TWO WIRE INTERFACE CONTROLLER

Specific Information for the ADSP-BF59x 12-1
Overview 12-2
Interface Overview 12-3
External Interface 12-4
Serial Clock Signal (SCL) 12-4
Serial Data Signal (SDA) 12-4
TWI Pins 12-5
Internal Interfaces 12-5
Description of Operation 12-6
TWI Transfer Protocols 12-6
Clock Generation and Synchronization 12-7
Bus Arbitration 12-8
Start and Stop Conditions 12-8
General Call Support 12-9
Fast Mode 12-10

Functional Description 12-10
General Setup 12-10
Slave Mode 12-11
Master Mode Clock Setup 12-12
Master Mode Transmit 12-12
Master Mode Receive 12-14
Repeated Start Condition 12-15
Transmit/Receive Repeated Start Sequence 12-15
Receive/Transmit Repeated Start Sequence 12-16
Clock Stretching 12-17
Clock Stretching During FIFO Underflow 12-17
Clock Stretching During FIFO Overflow 12-19
Clock Stretching During Repeated Start Condition 12-20
Programming Model 12-22
Register Descriptions 12-24
TWI CONTROL Register (TWI_CONTROL) 12-24
SCL Clock Divider Register (TWI_CLKDIV) 12-25
TWI Slave Mode Control Register (TWI_SLAVE_CTL) 12-26
TWI Slave Mode Address Register (TWI_SLAVE_ADDR) ... 12-28
TWI Slave Mode Status Register (TWI_SLAVE_STAT) 12-28
TWI Master Mode Control Register (TWI_MASTER_CTL) 12-29
TWI Master Mode Address Register (TWI_MASTER_ADDR) 12-32

TWI Master Mode Status Register (TWI_MASTER_STAT) 12-33
TWI FIFO Control Register (TWI_FIFO_CTL) 12-36
TWI FIFO Status Register (TWI_FIFO_STAT) 12-38
TWI FIFO Status 12-38
TWI Interrupt Mask Register (TWI_INT_MASK) 12-39
TWI Interrupt Status Register (TWI_INT_STAT) 12-40
TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8) 12-43
TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16) 12-43
TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8) 12-44
TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16) 12-45
Programming Examples 12-46
Master Mode Setup 12-46
Slave Mode Setup 12-50
Electrical Specifications 12-56
Unique Information for the ADSP-BF59x Processor 12-56

SPI-COMPATIBLE PORT CONTROLLER

Specific Information for the ADSP-BF59x 13-1
Overview 13-2
Features 13-2
Interface Overview 13-3

External Interface 13-4
SPI Clock Signal (SCK) 13-4
Master-Out, Slave-In (MOSI) Signal 13-5
Master-In, Slave-Out (MISO) Signal 13-5
SPI Slave Select Input Signal (SPISS) 13-6
SPI Slave Select Enable Output Signals 13-6
Slave Select Inputs 13-7
Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems 13-8
Internal Interfaces 13-10
DMA Functionality 13-10
Description of Operation 13-11
SPI Transfer Protocols 13-11
SPI General Operation 13-14
Clock Signals 13-15
Interrupt Output 13-16
Functional Description 13-16
Master Mode Operation (Non-DMA) 13-17
Transfer Initiation From Master (Transfer Modes) 13-18
Slave Mode Operation (Non-DMA) 13-19
Slave Ready for a Transfer 13-21
Programming Model 13-21
Beginning and Ending an SPI Transfer 13-21
Master Mode DMA Operation 13-23
Slave Mode DMA Operation 13-26

SPI Registers 13-33
SPI Baud Rate (SPI_BAUD) Register 13-34
SPI Control (SPI_CTL) Register 13-35
SPI Flag (SPI_FLG) Register 13-37
SPI Status (SPI_STAT) Register 13-38
Mode Fault Error (MODF) 13-39
Transmission Error (TXE) 13-40
Reception Error (RBSY) 13-41
Transmit Collision Error (TXCOL) 13-41
SPI Transmit Data Buffer (SPI_TDBR) Register 13-41
SPI Receive Data Buffer (SPI_RDBR) Register 13-42
SPI RDBR Shadow (SPI_SHADOW) Register 13-43
Programming Examples 13-43
Core-Generated Transfer 13-43
Initialization Sequence 13-44
Starting a Transfer 13-45
Post Transfer and Next Transfer 13-46
Stopping 13-46
DMA-Based Transfer 13-47
DMA Initialization Sequence 13-47
SPI Initialization Sequence 13-48
Starting a Transfer 13-49
Stopping a Transfer 13-50
Unique Information for the ADSP-BF59x Processor 13-52

SPORT CONTROLLER

Specific Information for the ADSP-BF59x 14-1
Overview 14-2
Features 14-2
Interface Overview 14-4
SPORT Pin/Line Terminations 14-8
Description of Operation 14-9
SPORT Disable 14-9
Setting SPORT Modes 14-10
Stereo Serial Operation 14-10
Multichannel Operation 14-14
Multichannel Enable 14-17
Frame Syncs in Multichannel Mode 14-18
The Multichannel Frame 14-19
Multichannel Frame Delay 14-20
Window Size 14-20
Window Offset 14-21
Other Multichannel Fields in SPORT_MCMC2 14-21
Channel Selection Register 14-22
Multichannel DMA Data Packing 14-23
Support for H.100 Standard Protocol 14-24
2× Clock Recovery Control 14-24
Functional Description 14-25
Clock and Frame Sync Frequencies 14-25

Maximum Clock Rate Restrictions 14-26
Word Length 14-27
Bit Order 14-27
Data Type 14-27
Companding 14-28
Clock Signal Options 14-29
Frame Sync Options 14-30
Framed Versus Unframed 14-30
Internal Versus External Frame Syncs 14-31
Active Low Versus Active High Frame Syncs 14-32
Sampling Edge for Data and Frame Syncs 14-32
Early Versus Late Frame Syncs (Normal Versus Alternate Timing) 14-34
Data Independent Transmit Frame Sync 14-36
Moving Data Between SPORTs and Memory 14-37
SPORT RX, TX, and Error Interrupts 14-37
Peripheral Bus Errors 14-38
Timing Examples 14-38
SPORT Registers 14-44
Register Writes and Effective Latency 14-45
SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers 14-46
SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers 14-51
Data Word Formats 14-55

SPORT Transmit Data (SPORT_TX) Register 14-56
SPORT Receive Data (SPORT_RX) Register 14-58
SPORT Status (SPORT_STAT) Register 14-60
SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers ... 14-61
SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers 14-62
SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers 14-63
SPORT Current Channel (SPORT_CHNL) Register 14-64
SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers 14-65
SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers 14-66
Programming Examples 14-67
SPORT Initialization Sequence 14-68
DMA Initialization Sequence 14-70
Interrupt Servicing 14-72
Starting a Transfer 14-73
Unique Information for the ADSP-BF59x Processor 14-73
Clock Gating Functionality 14-74
Modes of Operation 14-75
Gated Clock Mode 0 – SPORT Gated Clocks Without Using TIMERs 14-75
Gated Clock Mode 1 – SPORT Gated Clocks Using TIMERs 14-75

Programming Model 14-76

PARALLEL PERIPHERAL INTERFACE

Specific Information for the ADSP-BF59x 15-1
Overview 15-2
Features 15-2
Interface Overview 15-3
Description of Operation 15-4
Functional Description 15-5
ITU-R 656 Modes 15-5
ITU-R 656 Background 15-5
ITU-R 656 Input Modes 15-9
Entire Field 15-9
Active Video Only 15-10
Vertical Blanking Interval (VBI) only 15-10
ITU-R 656 Output Mode 15-11
Frame Synchronization in ITU-R 656 Modes 15-11
General-Purpose PPI Modes 15-12
Data Input (RX) Modes 15-14
No Frame Syncs 15-14
1, 2, or 3 External Frame Syncs 15-15
2 or 3 Internal Frame Syncs 15-16
Data Output (TX) Modes 15-16
No Frame Syncs 15-17
1 or 2 External Frame Syncs 15-17

1, 2, or 3 Internal Frame Syncs 15-18
Frame Synchronization in GP Modes 15-19
Modes With Internal Frame Syncs 15-19
Modes With External Frame Syncs 15-20
Programming Model 15-21
DMA Operation 15-22
PPI Registers 15-25
PPI Control Register (PPI_CONTROL) 15-25
PPI Status Register (PPI_STATUS) 15-29
PPI Delay Count Register (PPI_DELAY) 15-32
PPI Transfer Count Register (PPI_COUNT) 15-32
PPI Lines Per Frame Register (PPI_FRAME) 15-33
Programming Examples 15-34
Unique Information for the ADSP-BF59x Processor 15-37

SYSTEM RESET AND BOOTING

Overview 16-1
Reset and Power-up 16-3
Hardware Reset 16-4
Software Resets 16-5
Reset Vector 16-6
Servicing Reset Interrupts 16-6
Basic Booting Process 16-8
Block Headers 16-10
Block Code 16-12

DMA Code Field 16-12
Block Flags Field 16-14
Header Checksum Field 16-15
Header Sign Field 16-16
Target Address 16-16
Byte Count 16-17
Argument 16-17
Boot Host Wait (HWAIT) Feedback Strobe 16-18
Using HWAIT as Reset Indicator 16-19
Boot Termination 16-19
Single Block Boot Streams 16-20
Advanced Boot Techniques 16-21
Initialization Code 16-21
Quick Boot 16-25
Indirect Booting 16-26
Callback Routines 16-27
Error Handler 16-30
CRC Checksum Calculation 16-30
Load Functions 16-30
Calling the Boot Kernel at Runtime 16-32
Debugging the Boot Process 16-32
Boot Management 16-35
Booting a Different Application 16-35
Multi-DXE Boot Streams 16-36

Determining Boot Stream Start Addresses 16-37
Initialization Hook Routine 16-38
Specific Boot Modes 16-38
No Boot Mode 16-39
SPI Master Boot Modes 16-40
SPI Device Detection Routine 16-42
SPI Slave Boot Mode 16-44
PPI Boot Mode 16-47
UART Slave Mode Boot 16-49
L1 ROM Boot Mode 16-51
Reset and Booting Registers 16-51
Software Reset (SWRST) Register 16-52
System Reset Configuration (SYSCR) Register 16-54
Boot Code Revision Control (BK_REVISION) 16-55
Boot Code Date Code (BK_DATECODE) 16-56
Zero Word (BK_ZEROS) 16-57
Ones Word (BK_ONES) 16-58
Data Structures 16-58
ADI_BOOT_HEADER 16-59
ADI_BOOT_BUFFER 16-59
ADI_BOOT_DATA 16-59
dFlags Word 16-63
Callable ROM Functions for Booting 16-64
BFROM_FINALINIT 16-64

BFROM_PDMA 16-65
BFROM_MDMA 16-65
BFROM_SPIBOOT 16-66
BFROM_BOOTKERNEL 16-68
BFROM_CRC32 16-68
BFROM_CRC32POLY 16-69
BFROM_CRC32CALLBACK 16-70
BFROM_CRC32INITCODE 16-70
Programming Examples 16-71
System Reset 16-71
Exiting Reset to User Mode 16-72
Exiting Reset to Supervisor Mode 16-72
Initcode (Power Management Control) 16-73
XOR Checksum 16-75

SYSTEM DESIGN

Pin Descriptions 17-1
Managing Clocks 17-1
Managing Core and System Clocks 17-2
Configuring and Servicing Interrupts 17-2
Data Delays, Latencies and Throughput 17-2
Bus Priorities 17-3
High-Frequency Design Considerations 17-3
Signal Integrity 17-3
Decoupling Capacitors and Ground Planes 17-4

Test Point Access 17-6
Oscilloscope Probes 17-6
Recommended Reading 17-7
Resetting the Processor 17-8
Recommendations for Unused Pins 17-8
Programmable Outputs 17-9
Voltage Regulation Interface 17-9

SYSTEM MMR ASSIGNMENTS

Processor-Specific Memory Registers A-2
Core Timer Registers A-3
System Reset and Interrupt Control Registers A-3
DMA/Memory DMA Control Registers A-4
Ports Registers A-7
Timer Registers A-9
Watchdog Timer Registers A-11
Dynamic Power Management Registers A-11
PPI Registers A-12
SPI Controller Registers A-12
SPORT Controller Registers A-14
SPORT Clock Gating Register A-17
UART Controller Registers A-18
TWI Registers A-19

TEST FEATURES

JTAG Standard B-1
Boundary-Scan Architecture B-2
Instruction Register B-4
Public Instructions B-5
EXTEST – Binary Code 00000 ..... B-6
SAMPLE/PRELOAD – Binary Code 10000 ..... B-6
BYPASS – Binary Code 11111 ..... B-6
Boundary-Scan Register B-7

INDEX

PREFACE

Thank you for purchasing and developing systems using an enhanced Blackfin® processor from Analog Devices.

Purpose of This Manual

The *ADSP-BF59x Blackfin Processor Hardware Reference* provides architectural information about the ADSP-BF59x processors. This hardware reference provides the main architectural information about these processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support. For programming information, see the *Blackfin Processor Programming Reference.* For timing, electrical, and package specifications, see the *ADSP-BF592 Blackfin Processor Data Sheet*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as hardware reference and programming reference manuals, that describe their target architecture.

Manual Contents

This manual consists of one volume:

- Chapter 1, “Introduction”
  Provides a high level overview of the processor, including peripherals, power management, and development tools.
- Chapter 2, “Memory”
  Describes processor-specific memory topics, including L1memories and processor-specific memory MMRs.
- Chapter 3, “Chip Bus Hierarchy”
  Describes on-chip buses, including how data moves through the system.
- Chapter 4, “System Interrupts”
  Describes the system peripheral interrupts, including setup and clearing of interrupt requests.
- Chapter 5, “Direct Memory Access”
  Describes the peripheral DMA and Memory DMA controllers. Includes performance, software management of DMA, and DMA errors.
- Chapter 6, “Dynamic Power Management”
  Describes the clocking, including the PLL, and the dynamic power management controller.
- Chapter 7, “General-Purpose Ports”
  Describes the general-purpose I/O ports, including the structure of each port, multiplexing, configuring the pins, and generating interrupts.
- Chapter 8, “General-Purpose Timers”
  Describes the eight general-purpose timers.

- Chapter 9, “Core Timer”
  Describes the core timer.
- Chapter 10, “Watchdog Timer”
  Describes the watchdog timer.
- Chapter 11, “UART Port Controllers”
  Describes the Universal Asynchronous Receiver/Transmitter port that converts data between serial and parallel formats. The UART supports the half-duplex IrDA® SIR protocol as a mode-enabled feature.
- Chapter 12, “Two Wire Interface Controller”
  Describes the Two Wire Interface (TWI) controller, which allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification version 2.1* dated January 2000.
- Chapter 13, “SPI-Compatible Port Controller”
  Describes the Serial Peripheral Interface (SPI) port that provides an I/O interface to a variety of SPI compatible peripheral devices.
- Chapter 14, “SPORT Controller”
  Describes the independent, synchronous Serial Port Controller which provides an I/O interface to a variety of serial peripheral devices.
- Chapter 15, “Parallel Peripheral Interface”
  Describes the Parallel Peripheral Interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and is used for digital video and data converter applications.
- Chapter 16, “System Reset and Booting”
  Describes the booting methods, booting process and specific boot modes for the processor.

- Chapter 17, "System Design"
  Describes how to use the processor as part of an overall system. It includes information about bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.
- Appendix A, "System MMR Assignments"
  Lists the memory-mapped registers included in this manual, their addresses, and cross-references to text.
- Appendix B, "Test Features"
  Describes test features for the processor, discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.

This hardware reference is a companion document to the *Blackfin Processor Programming Reference*.

What's New in This Manual

This revision (1.0) is the second release of the *ADSP-BF59x Blackfin Processor Hardware Reference*. Minor typographical errors have been corrected in this revision.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at
  `http://www.analog.com/processors/technical_support`
- E-mail tools questions to
  `processor.tools.support@analog.com`

- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

Registration for MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. Click **Register** to use this site. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit http://ez.analog.com to sign up.

Social Networking Web Sites

You can now follow Analog Devices SHARC development on Twitter and LinkedIn. To access:

- Twitter: http://twitter.com/ADISHARC
- LinkedIn: Network with the LinkedIn group, Analog Devices SHARC: http://www.linkedin.com

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++®.

Blackfin (ADSP-BFxxx) Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin families ADSP-BF51x, ADSP-BF52x, ADSP-BF53x, ADSP-BF54x, ADSP-BF59x, and ADSP-BF561 processors.

TigerSHARC® (ADSP-TSxxx) Processors

The name *TigerSHARC* refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC families: ADSP-TS101 and ADSP-TS20x.

SHARC® (ADSP-21xxx) Processors

The name *SHARC* refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC families: ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-214xx.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

The Analog Devices Web site, www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit www.myanalog.com to sign up. If you are already a registered user, just log on. Your user name is your e-mail address.

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Microsoft help format
.htm or .html	Dinkum Abridged C++ library and FLEXnet License Tools software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in PDF format. Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin®, SHARC, TigerSHARC®, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.com/processors/manuals, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.

Example	Description
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for the devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION

The ADSP-BF59x processors are members of the Blackfin processor family that offer significant high performance and low power features while retaining their ease-of-use benefits.

This hardware reference is a companion document to the *Blackfin Processor Programming Reference*.

General Description of Processor

The ADSP-BF59x processor is a member of the Blackfin® family of products, incorporating the Analog Devices/Intel Micro Signal Architecture (MSA). Blackfin processors combine a dual-MAC state-of-the-art signal processing engine, the advantages of a clean, orthogonal RISC-like microprocessor instruction set, and single-instruction, multiple-data (SIMD) multimedia capabilities into a single instruction-set architecture.

The ADSP-BF59x processor is completely code compatible with other Blackfin processors. ADSP-BF59x processors offer performance up to 400 MHz and reduced static power consumption. The processor features are shown in Table 1-1.

By integrating a rich set of industry-leading system peripherals and memory, Blackfin processors are the platform of choice for next-generation applications that require RISC-like programmability, multimedia support, and leading-edge signal processing in one integrated package.

Table 1-1. Processor Features

Feature		ADSP-BF592
Timer/Counters with PWM		3
SPORTs		2
SPIs		2
UART		1
Parallel Peripheral Interface		1
TWI		1
GPIOs		32
Memory (bytes)	L1 Instruction SRAM	32K
	L1 Instruction ROM	64K
	L1 Data SRAM	32K
	L1 Scratchpad	4K
	L3 Boot ROM	4K
Maximum Instruction Rate[1]		400 MHz
Maximum System Clock Speed		100 MHz
Package Options		64-Lead LFCSP

1 Maximum instruction rate is not available with every possible SCLK selection.

Portable Low-Power Architecture

Blackfin processors provide world-class power management and performance. They are produced with a low-power and low-voltage design methodology and feature on-chip dynamic power management, which provides the ability to vary both the voltage and frequency of operation to significantly lower overall power consumption. This capability can result in a substantial reduction in power consumption, compared with just

varying the frequency of operation. This allows longer battery life for portable appliances.

Peripherals

The processor system peripherals include:

- Two memory-to-memory DMAs
- Event handler with 28 interrupt inputs
- 9 peripheral DMAs
- 32 General-Purpose I/Os (GPIOs)
- Three 32-bit timer/counters with PWM support
- 32-bit core timer
- On-chip PLL capable of 5× to 64× frequency multiplication
- Debug/JTAG interface
- Parallel Peripheral Interface (PPI), supporting ITU-R 656 video data formats
- Two Serial Peripheral Interface (SPI)-compatible ports
- Two-Wire Interface (TWI) controller
- Two dual-channel, full-duplex synchronous Serial Ports (SPORTs), supporting eight stereo I^2S channels
- One UART with IrDA® support

These peripherals are connected to the core via several high bandwidth buses, as shown in Figure 1-1.

Most of the peripherals are supported by a flexible DMA structure. There are also two separate memory DMA channels dedicated to data transfers between the processor's memory spaces. Multiple on-chip buses provide enough bandwidth to keep the processor core running even when there is also activity on all of the on-chip and external peripherals.

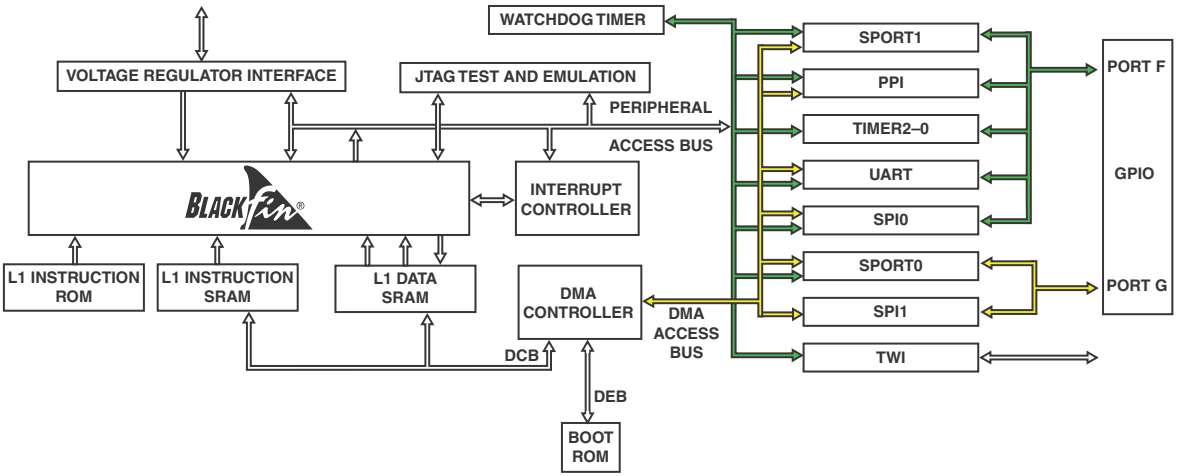


Figure 1-1. ADSP-BF59x Processor Block Diagram

Memory Architecture

The Blackfin processor architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory,

and larger, lower cost and lower performance off-chip memory systems. Table 1-2 shows the memory for the ADSP-BF59x processors.

Table 1-2. Memory Configurations

Type of Memory	ADSP-BF59x
Instruction SRAM	32K byte
Instruction ROM	64K byte
Data SRAM	32K byte
Data scratchpad SRAM	4K byte
L3 Boot ROM	4K byte
Total	136K byte

Internal Memory

The processor has four blocks of on-chip memory that provide high bandwidth access to the core:

- L1 instruction SRAM memory. This memory is accessed at full processor speed.
- L1 data SRAM memory. This memory block is accessed at full processor speed.
- L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM.
- L1 instruction ROM memory, accessed at full processor speed.

I/O Memory Space

Blackfin processors do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. Control registers for on-chip I/O devices are mapped into memory-mapped registers (MMRs) at addresses near the top of the 4G byte address space. These are separated

into two smaller blocks: one contains the control MMRs for all core functions and the other contains the registers needed for setup and control of the on-chip peripherals outside of the core. The MMRs are accessible only in supervisor mode. They appear as reserved space to on-chip peripherals.

DMA Support

The processor has a DMA controller which supports automated data transfers with minimal overhead for the core. DMA transfers can occur between the internal memories and any of its DMA-capable peripherals. DMA-capable peripherals include the SPORTs, SPI ports, UART, and PPI. Each individual DMA-capable peripheral has at least one dedicated DMA channel.

The DMA controller supports both one-dimensional (1D) and two-dimensional (2-D) DMA transfers. DMA transfer initialization can be implemented from registers or from sets of parameters called descriptor blocks.

The 2-D DMA capability supports arbitrary row and column sizes up to 64K elements by 64K elements, and arbitrary row and column step sizes up to +/- 32K elements. Furthermore, the column step size can be less than the row step size, allowing implementation of interleaved data-streams. This feature is especially useful in video applications where data can be de-interleaved on the fly.

Examples of DMA types supported include:

- A single, linear buffer that stops upon completion
- A circular, auto-refreshing buffer that interrupts on each full or fractionally full buffer

- 1-D or 2-D DMA using a linked list of descriptors
- 2-D DMA using an array of descriptors specifying only the base DMA address within a common page

In addition to the dedicated peripheral DMA channels, there are two separate pairs of memory DMA channels provided for transfers between the various memories of the system. Memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

General-Purpose I/O (GPIO)

The ADSP-BF59x processors have 32 bi-directional, general-purpose I/O (GPIO) pins allocated across two separate GPIO modules—PORTFIO, and PORTGIO, associated with port F and port G, respectively. Port J does not provide GPIO functionality. Each GPIO-capable pin shares functionality with other ADSP-BF59x processor peripherals via a multiplexing scheme; however, the GPIO functionality is the default state of the device upon powerup. Neither GPIO output or input drivers are active by default. Each general-purpose port pin can be individually controlled by manipulation of the port control, status, and interrupt registers:

- GPIO direction control register – Specifies the direction of each individual GPIO pin as input or output.
- GPIO control and status registers – The ADSP-BF59x processors employ a “write one to modify” mechanism that allows any combination of individual GPIO pins to be modified in a single instruction, without affecting the level of any other GPIO pins. Four control registers are provided. One register is written in order to set pin values, one register is written in order to clear pin values, one register is written in order to toggle pin values, and one register is written in order to specify a pin value. Reading the GPIO status register allows software to interrogate the sense of the pins.

- GPIO interrupt mask registers – The two GPIO interrupt mask registers allow each individual GPIO pin to function as an interrupt to the processor. Similar to the two GPIO control registers that are used to set and clear individual pin values, one GPIO interrupt mask register sets bits to enable interrupt function, and the other GPIO interrupt mask register clears bits to disable interrupt function. GPIO pins defined as inputs can be configured to generate hardware interrupts, while output pins can be triggered by software interrupts.
- GPIO interrupt sensitivity registers – The two GPIO interrupt sensitivity registers specify whether individual pins are level- or edge-sensitive and specify—if edge-sensitive—whether just the rising edge or both the rising and falling edges of the signal are significant. One register selects the type of sensitivity, and one register selects which edges are significant for edge-sensitivity.

Two-Wire Interface

The Two-Wire Interface (TWI) is compatible with the widely used I^2C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth, the TWI controller can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the I^2C bus protocol. The *Philips I^2C Bus Specification version 2.1* covers many variants of I^2C. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master data arbitration

- 7-bit addressing
- 100K bits/second and 400K bit/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up
- Input filter for spike suppression
- Serial camera control bus support as specified in the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*

Parallel Peripheral Interface

The processor provides a Parallel Peripheral Interface (PPI) that can connect directly to parallel A/D and D/A converters, ITU-R 601/656 video encoders and decoders, and other general-purpose peripherals. The PPI consists of a dedicated input clock pin and three multiplexed frame sync pins. The input clock supports parallel data rates up to half the system clock rate.

In ITU-R 656 modes, the PPI receives and parses a data stream of 8-bit or 10-bit data elements. On-chip decode of embedded preamble control and synchronization information is supported.

Three distinct ITU-R 656 modes are supported:

- Active video only - The PPI does not read in any data between the End of Active Video (EAV) and Start of Active Video (SAV) preamble symbols, or any data present during the vertical blanking intervals. In this mode, the control byte sequences are not stored to memory; they are filtered by the PPI.
- Vertical blanking only - The PPI only transfers Vertical Blanking Interval (VBI) data, as well as horizontal blanking information and control byte sequences on VBI lines.
- Entire field - The entire incoming bitstream is read in through the PPI. This includes active video, control preamble sequences, and ancillary data that may be embedded in horizontal and vertical blanking intervals.

Though not explicitly supported, ITU-R 656 output functionality can be achieved by setting up the entire frame structure (including active video, blanking, and control information) in memory and streaming the data out the PPI in a frame sync-less mode. The processor's 2-D DMA features facilitate this transfer by allowing the static frame buffer (blanking and control codes) to be placed in memory once, and simply updating the active video information on a per-frame basis.

The general-purpose modes of the PPI are intended to suit a wide variety of data capture and transmission applications. The modes are divided into four main categories, each allowing up to 16 bits of data transfer per `PPI_CLK` cycle:

- Data receive with internally generated frame syncs
- Data receive with externally generated frame syncs
- Data transmit with internally generated frame syncs
- Data transmit with externally generated frame syncs

These modes support ADC/DAC connections, as well as video communication with hardware signalling. Many of the modes support more than one level of frame synchronization. If desired, a programmable delay can be inserted between assertion of a frame sync and reception/transmission of data.

SPORT Controllers

The processor incorporates two dual-channel synchronous serial ports (SPORT0 and SPORT1) for serial and multiprocessor communications. The SPORTs support these features:

- Bidirectional, I^2S capable operation

  Each SPORT has two sets of independent transmit and receive pins, which enable eight channels of I^2S stereo audio.

- Buffered (eight-deep) transmit and receive ports

  Each port has a data register for transferring data words to and from other processor components and shift registers for shifting data in and out of the data registers.

- Clocking

  Each transmit and receive port can either use an external serial clock or can generate its own in a wide range of frequencies.

- Word length

  Each SPORT supports serial data words from 3 to 32 bits in length, transferred in most significant bit first or least significant bit first format.

- Framing

  Each transmit and receive port can run with or without frame sync signals for each data word. Frame sync signals can be generated internally or externally, active high or low, and with either of two pulse widths and early or late frame sync.

- Companding in hardware

  Each SPORT can perform A-law or µ-law companding according to ITU recommendation G.711. Companding can be selected on the transmit and/or receive channel of the SPORT without additional latencies.

- DMA operations with single cycle overhead

  Each SPORT can automatically receive and transmit multiple buffers of memory data. The processor can link or chain sequences of DMA transfers between a SPORT and memory.

- Interrupts

  Each transmit and receive port generates an interrupt upon completing the transfer of a data word or after transferring an entire data buffer or buffers through DMA.

- Multichannel capability

  Each SPORT supports 128 channels out of a 1024-channel window and is compatible with the H.100, H.110, MVIP-90, and HMVIP standards.

Serial Peripheral Interface (SPI) Ports

The processor has two SPI-compatible ports that enable the processor to communicate with multiple SPI-compatible devices.

Each SPI interface uses three pins for transferring data: two data pins and a clock pin. An SPI chip select input pin lets other SPI devices select the processor, and several SPI chip select output pins let the processor select other SPI devices. The SPI select pins are reconfigured general-purpose I/O pins. Using these pins, the SPI port provides a full-duplex, synchronous serial interface, which supports both master and slave modes and multimaster environments.

Each SPI port's baud rate and clock phase/polarities are programmable, and it has an integrated DMA controller, configurable to support either transmit or receive datastreams. The SPI's DMA controller can only service unidirectional accesses at any given time.

During transfers, the SPI port simultaneously transmits and receives by serially shifting data in and out of its two serial data lines. The serial clock line synchronizes the shifting and sampling of data on the two serial data lines.

Timers

There are three general-purpose programmable timer units in the processor. Eight timers have an external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events. These timer units can be synchronized to an external clock input connected to the `TMRCLK`/`PPI_CLK` pin or to the internal `SCLK`.

The timer units can be used in conjunction with the UARTs to measure the width of the pulses in the datastream to provide an autobaud detect function for a serial channel.

The timers can generate interrupts to the processor core to provide periodic events for synchronization, either to the processor clock or to a count of external signals.

In addition to the eight general-purpose programmable timers, a 9th timer is also provided. This extra timer is clocked by the internal processor clock and is typically used as a system tick clock for generation of operating system periodic interrupts.

UART Port

The processor provides one Universal Asynchronous Receiver/Transmitter (UART) port, which is fully compatible with PC-standard UARTs. The UART port provides a simplified UART interface to other peripherals or hosts, providing half-duplex, DMA-supported, asynchronous transfers of serial data. The UART port includes support for 5 to 8 data bits; 1 or 2 stop bits; and none, even, or odd parity. The UART port supports two modes of operation:

- Programmed I/O

  The processor sends or receives data by writing or reading I/O-mapped UART registers. The data is double buffered on both transmit and receive.

- Direct Memory Access (DMA)

  The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. The UART has two dedicated DMA channels, one for transmit and one for receive. These DMA channels have lower priority than most DMA channels because of their relatively low service rates.

The UART's baud rate, serial data format, error code generation and status, and interrupts can be programmed to support:

- Wide range of bit rates
- Data formats from 7 to 12 bits per frame
- Generation of maskable interrupts to the processor by both transmit and receive operations

In conjunction with the general-purpose timer functions, autobaud detection is supported.

The capabilities of the UART port is further extended with support for the Infrared Data Association (IrDA®) Serial Infrared Physical Layer Link Specification (SIR) protocol.

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system availability by forcing the processor to a known state through generation of a hardware reset, nonmaskable interrupt (NMI), or general-purpose interrupt, if the timer expires before being reset by software. The programmer initializes the count value of the timer, enables the appropriate interrupt, then enables the timer. Thereafter, the software must reload the counter before it counts to zero from the programmed value. This protects the system from remaining in an unknown state where software that would normally reset the timer has stopped running due to an external noise condition or software error.

If configured to generate a hardware reset, the watchdog timer resets both the CPU and the peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the watchdog control register.

The timer is clocked by the system clock (SCLK), at a maximum frequency of f_{SCLK}.

Clock Signals

The processor can be clocked by an external crystal, a sine wave input, or a buffered, shaped clock derived from an external clock oscillator.

This external clock connects to the processor's CLKIN pin. The CLKIN input cannot be halted, changed, or operated below the specified frequency during normal operation. This clock signal should be a TTL-compatible signal.

The core clock (CCLK) and system peripheral clock (SCLK) are derived from the input clock (CLKIN) signal. An on-chip Phase Locked Loop (PLL) is capable of multiplying the CLKIN signal by a user-programmable (5× to 64×) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 6×, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.

All on-chip peripherals are clocked by the system clock (SCLK). The system clock frequency is programmable by means of the SSEL[3:0] bits of the PLL_DIV register.

Dynamic Power Management

The processor provides four operating modes, each with a different performance/power profile. In addition, dynamic power management provides the control functions to dynamically alter the processor core supply voltage to further reduce power dissipation. Control of clocking to each of the peripherals also reduces power consumption.

Full-On Mode (Maximum Performance)

In the full-on mode, the PLL is enabled, not bypassed, providing the maximum operational frequency. This is the normal execution state in which maximum performance can be achieved. The processor core and all enabled peripherals run at full speed.

Active Mode (Moderate Power Savings)

In the active mode, the PLL is enabled, but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. In this mode, the CLKIN to VCO multiplier ratio can be changed, although the changes are not realized until the full on mode is entered. DMA access is available to appropriately configured L1 memories.

In the active mode, it is possible to disable the PLL through the PLL control register (PLL_CTL). If disabled, the PLL must be re-enabled before transitioning to the full on or sleep modes.

Sleep Mode (High Power Savings)

The sleep mode reduces power dissipation by disabling the clock to the processor core (CCLK). The PLL and system clock (SCLK), however, continue to operate in this mode. Typically an external event will wake up the processor. When in the sleep mode, assertion of any interrupt causes the processor to sense the value of the bypass bit (BYPASS) in the PLL control register (PLL_CTL). If bypass is disabled, the processor transitions to the full on mode. If bypass is enabled, the processor transitions to the active mode.

When in the sleep mode, system DMA access to L1 memory is not supported.

Deep Sleep Mode (Maximum Power Savings)

The deep sleep mode maximizes dynamic power savings by disabling the processor core and synchronous system clocks (CCLK and SCLK). Asynchronous systems may still be running, but cannot access internal resources or external memory. This powered-down mode can only be exited by assertion of the reset interrupt or by an wakeup input. When in deep sleep mode, a wakeup input interrupt causes the processor to transition to the active mode. Assertion of $\overline{\text{RESET}}$ while in deep sleep mode causes the processor to transition to the full on mode.

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such.

Instruction Set Description

The Blackfin processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. Refer to the *Blackfin Processor Programming Reference* for detailed information. The instructions have been specifically tuned to provide a flexible, densely encoded instruction set that compiles to a very small final memory size. The instruction set also provides fully featured multifunction instructions that allow the programmer to use many of the processor core resources in a single instruction. Coupled with many features more often seen on microcontrollers, this instruction set is very efficient when compiling C and C++ source code. In addition, the architecture supports both user (algorithm/application code) and supervisor (O/S kernel, device drivers, debuggers, ISRs) modes of operation, allowing multiple levels of access to core resources.

The assembly language, which takes advantage of the processor's unique architecture, offers these advantages:

- Embedded 16/32-bit microcontroller features, such as arbitrary bit and bit field manipulation, insertion, and extraction; integer operations on 8-, 16-, and 32-bit data types; and separate user and supervisor stack pointers
- Seamlessly integrated DSP/CPU features optimized for both 8-bit and 16-bit operations
- A multi-issue load/store modified Harvard architecture, which supports two 16-bit MAC or four 8-bit ALU + two load/store + two pointer updates per cycle
- All registers, I/O, and memory mapped into a unified 4G byte memory space, providing a simplified programming model

Code density enhancements include intermixing of 16- and 32-bit instructions with no mode switching or code segregation. Frequently used instructions are encoded in 16 bits.

Development Tools

The processor is supported with a complete set of CROSSCORE® software and hardware development tools, including Analog Devices emulators and the VisualDSP++® development environment. The same emulator hardware that supports other Analog Devices products also fully emulates the Blackfin processor family.

The VisualDSP++ project management environment lets programmers develop and debug an application. This environment includes an easy-to-use assembler that is based on an algebraic syntax, an archiver (librarian/library builder), a linker, a loader, a cycle-accurate instruction-level simulator, a C/C++ compiler, and a C/C++ runtime library that

includes DSP and mathematical functions. A key point for these tools is C/C++ code efficiency. The compiler has been developed for efficient translation of C/C++ code to Blackfin processor assembly. The Blackfin processor has architectural features that improve the efficiency of compiled C/C++ code.

Debugging both C/C++ and assembly programs with the VisualDSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information)
- Insert breakpoints
- Set conditional breakpoints on registers, memory, and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution
- Fill, dump, and graphically plot the contents of memory
- Perform source level debugging
- Create custom debugger windows

The VisualDSP++ Integrated Development and Debugging Environment (IDDE) lets programmers define and manage software development. Its dialog boxes and property pages let programmers configure and manage all development tools, including color syntax highlighting in the VisualDSP++ editor. These capabilities permit programmers to:

- Control how the development tools process inputs and generate outputs
- Maintain a one-to-one correspondence with the tool's command-line switches

The VisualDSP++ Kernel (VDK) incorporates scheduling and resource management tailored specifically to address the memory and timing constraints of DSP programming. These capabilities enable engineers to develop code more effectively, eliminating the need to start from the very beginning, when developing new application code. The VDK features include threads, critical and unscheduled regions, semaphores, events, and device flags. The VDK also supports priority-based, pre-emptive, cooperative and time-sliced scheduling approaches. In addition, the VDK was designed to be scalable. If the application does not use a specific feature, the support code for that feature is excluded from the target system.

Because the VDK is a library, a developer can decide whether to use it or not. The VDK is integrated into the VisualDSP++ development environment but can also be used with standard command-line tools. The VDK development environment assists in managing system resources, automating the generation of various VDK-based objects, and visualizing the system state during application debug.

Analog Devices emulators use the IEEE 1149.1 JTAG test access port of the processor to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Nonintrusive in-circuit emulation is assured by the use of the processor's JTAG interface—the emulator does not affect target system loading or timing.

In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processor family. Hardware tools include the ADSP-BF59x EZ-KIT Lite standalone evaluation/development cards. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 MEMORY

This chapter discusses memory population specific to the ADSP-BF59x processors. Functional memory architecture is described in the *Blackfin Processor Programming Reference*.

Note that the ADSP-BF59x processors do not have L1 instruction cache or data cache. For ADSP-BF59x processors, disregard those portions of the *Blackfin Processor Programming Reference* that pertain to cache.

Memory Architecture

Figure 2-1 on page 2-2 provides an overview of the ADSP-BF59x processor system memory map. For a detailed discussion of how to use them, see the *Blackfin Processor Programming Reference*.

Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

The upper portion of internal memory space is allocated to the core and system MMRs. Accesses to this area are allowed only when the processor is in supervisor or emulation mode (see the Operating Modes and States chapter of the *Blackfin Processor Programming Reference*).

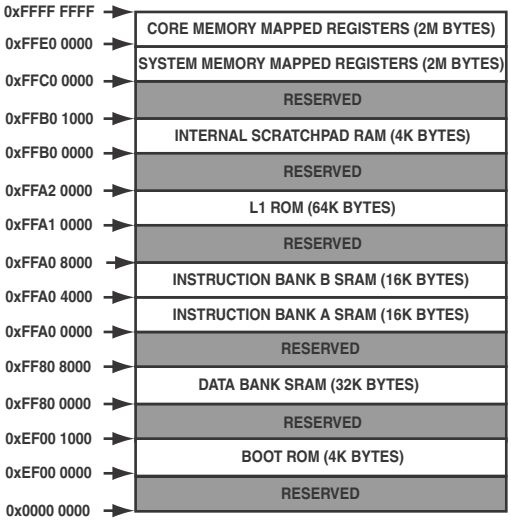


Figure 2-1. ADSP-BF59x Memory Map

L1 Instruction SRAM

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32-, or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

Table 2-1 lists the memory start locations of the L1 instruction SRAM subbanks.

Table 2-1. L1 Instruction Memory Subbanks

Memory Bbank	Memory Subbank	Memory Start Location for ADSP-BF59x Processors
Instruction Bank A	0	0xFFA0 0000
Instruction Bank A	1	0xFFA0 1000

Table 2-1. L1 Instruction Memory Subbanks (Continued)

Memory Bbank	Memory Subbank	Memory Start Location for ADSP-BF59x Processors
Instruction Bank A	2	0xFFA0 2000
Instruction Bank A	3	0xFFA0 3000
Instruction Bank B	0	0xFFA0 4000
Instruction Bank B	1	0xFFA0 5000
Instruction Bank B	2	0xFFA0 6000
Instruction Bank B	3	0xFFA0 7000

L1 Instruction ROM

The 64K byte L1 instruction ROM consists of a single 64K byte bank of read-only memory. The instruction ROM is typically read by the processor to acquire instructions for execution, but contents of instruction ROM may also be read using the DTEST_COMMAND and DTEST_DATA registers. Attempts to write ROM using the DTEST_COMMAND and DTEST_DATA registers fail *without* any errors or exceptions signaled by hardware. DMA access of instruction ROM is not possible.

L1 Data SRAM

Table 2-2 shows how the subbank organization is mapped into memory.

Table 2-2. L1 Data Memory SRAM Subbank Start Addresses

Memory Bank and Subbank	ADSP-BF59x Processors
Data Bank A, Subbank 0	0xFF80 0000
Data Bank A, Subbank 1	0xFF80 1000
Data Bank A, Subbank 2	0xFF80 2000

Table 2-2. L1 Data Memory SRAM Subbank Start Addresses (Continued)

Memory Bank and Subbank	ADSP-BF59x Processors
Data Bank A, Subbank 3	0xFF80 3000
Data Bank A, Subbank 4	0xFF80 4000
Data Bank A, Subbank 5	0xFF80 5000
Data Bank A, Subbank 6	0xFF80 6000
Data Bank A, Subbank 7	0xFF80 7000

Boot ROM

A 4K byte area of internal memory space is occupied by the boot ROM, starting from address 0xEF00 0000. This 16-bit boot ROM is not part of the L1 memory module. Read accesses take one SCLK cycle and no wait states are required. The read-only memory can be read by the core as well as by DMA. The boot ROM not only contains boot-strap loader code, it also provides some subfunctions that are user-callable at runtime. For more information, see "System Reset and Booting" in Chapter 16, System Reset and Booting.

External Memory

Aside from the Boot ROM, which sits in External Memory space, there is no additional external memory address space on the processor.

Processor-Specific MMRs

The complete set of memory-related MMRs is described in the *Blackfin Processor Programming Reference*. Several MMRs have bit definitions specific to the processors described in this manual. These registers are described in the following sections.

DTEST_COMMAND Register

When the data test command register (DTEST_COMMAND) is written to, L1 memory is accessed, and the data is transferred through the data test data registers (DTEST DATA[1:0]). This register is shown in Figure 2-2.

The data/instruction access bit allows direct access via the DTEST_COMMAND MMR to L1 instruction SRAM.

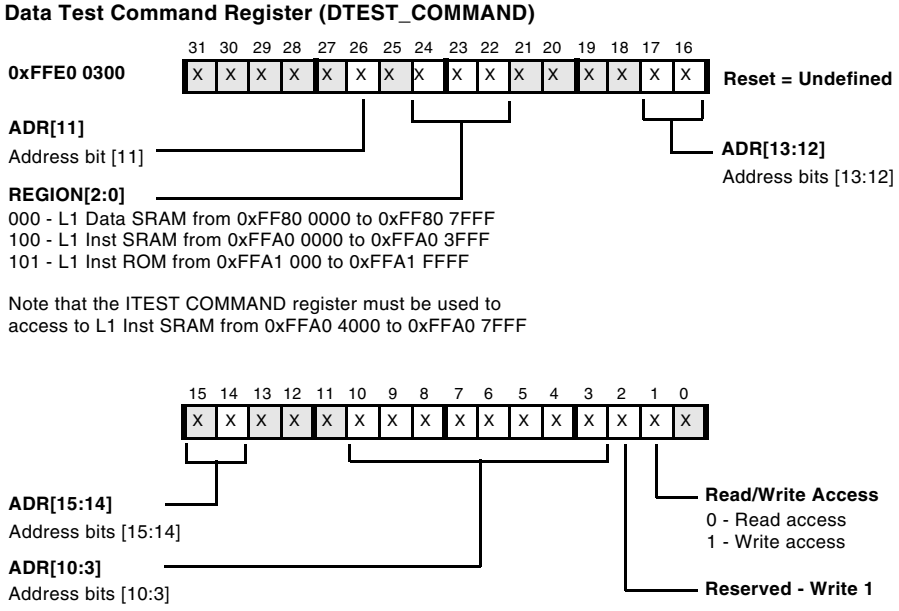

Data Test Command Register (DTEST_COMMAND)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0xFFE0 0300
X X X X X X X X X X X X X X X X
Reset = Undefined
ADR[11]
Address bit [11]
REGION[2:0]
000 - L1 Data SRAM from 0xFF80 0000 to 0xFF80 7FFF
100 - L1 Inst SRAM from 0xFFA0 0000 to 0xFFA0 3FFF
101 - L1 Inst ROM from 0xFFA1 000 to 0xFFA1 FFFF
Note that the ITEST COMMAND register must be used to access to L1 Inst SRAM from 0xFFA0 4000 to 0xFFA0 7FFF
ADR[13:12]
Address bits [13:12]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
ADR[15:14]
Address bits [15:14]
ADR[10:3]
Address bits [10:3]
Read/Write Access
0 - Read access
1 - Write access
Reserved - Write 1

Figure 2-2. Data Test Command Register

ITEST_COMMAND Register

The instruction test command register (ITEST_COMMAND), shown in Figure 2-3, contains control bits for the L1 data memory.

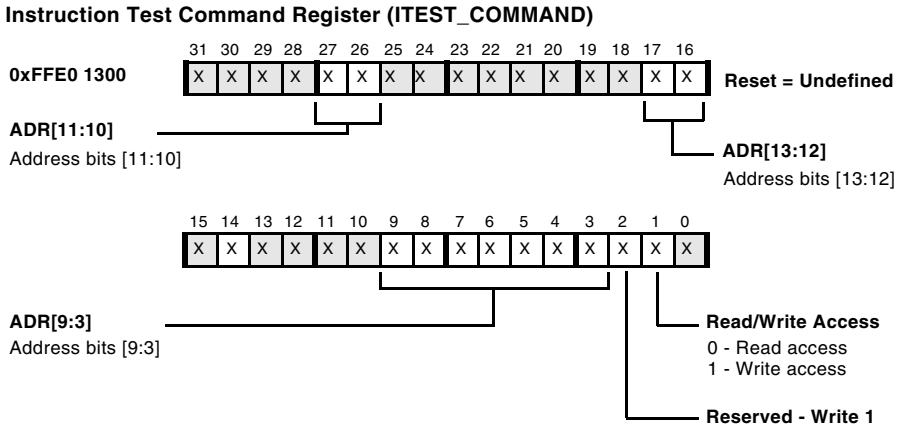

Instruction Test Command Register (ITEST_COMMAND)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0xFFE0 1300
X X X X X X X X X X X X X X X X
Reset = Undefined
ADR[11:10]
Address bits [11:10]
ADR[13:12]
Address bits [13:12]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
ADR[9:3]
Address bits [9:3]
Read/Write Access
0 - Read access
1 - Write access
Reserved - Write 1

Figure 2-3. Instruction Test Command Register

This register may be used to gain access to the 16K bytese of L1 instruction SRAM from address 0xFFA04000 to address 0xFFA07FFF. All other regions of L1 memory—both data and instruction—are accessed using the DTEST_COMMAND register.

DMEM_CONTROL Register

The data memory control register (DMEM_CONTROL), shown in Figure 2-4, contains control bits for the L1 data memory.

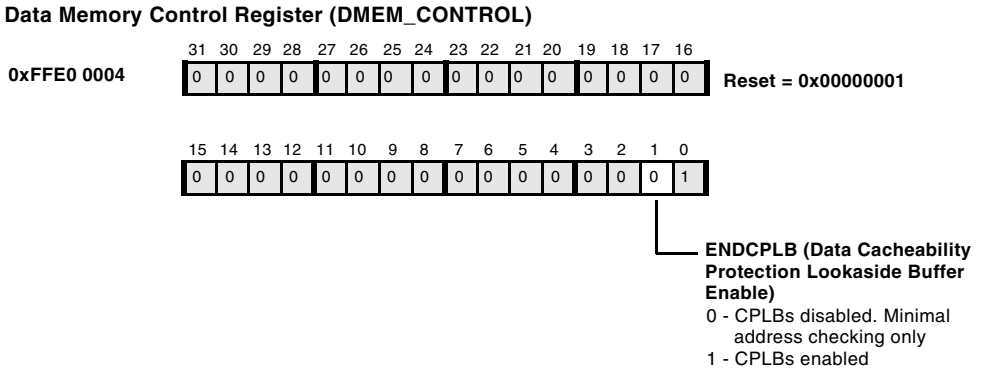

Data Memory Control Register (DMEM_CONTROL)
0xFFE0 0004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x00000001
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
ENDCPLB (Data Cacheability Protection Lookaside Buffer Enable)
0 - CPLBs disabled. Minimal address checking only
1 - CPLBs enabled

Figure 2-4. Data Memory Control Register

IMEM_CONTROL Register

The instruction memory control register (IMEM_CONTROL), shown in Figure 2-5, contains control bits for the L1 instruction memory.

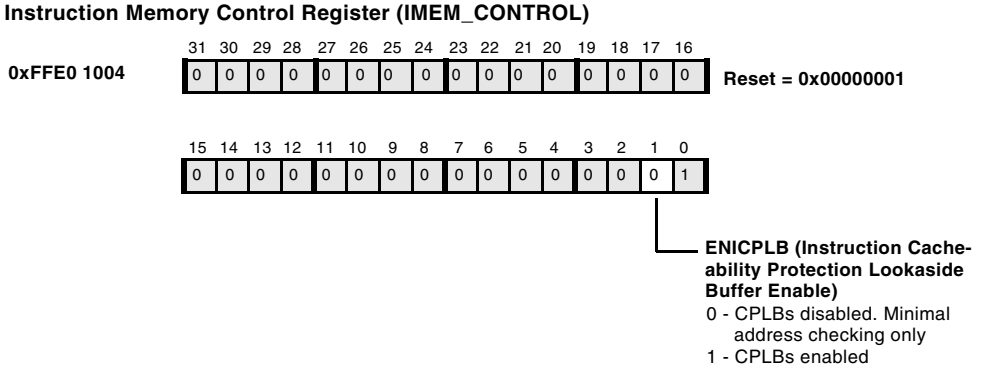

Instruction Memory Control Register (IMEM_CONTROL)
0xFFE0 1004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x00000001
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
ENICPLB (Instruction Cacheability Protection Lookaside Buffer Enable)
0 - CPLBs disabled. Minimal address checking only
1 - CPLBs enabled

Figure 2-5. Instruction Memory Control Register

DCPLB_DATAx Registers

The data CPLB data registers (DCPLB_DATAx), shown in Figure 2-6, contain CPLB control bits for the L1 data memory.

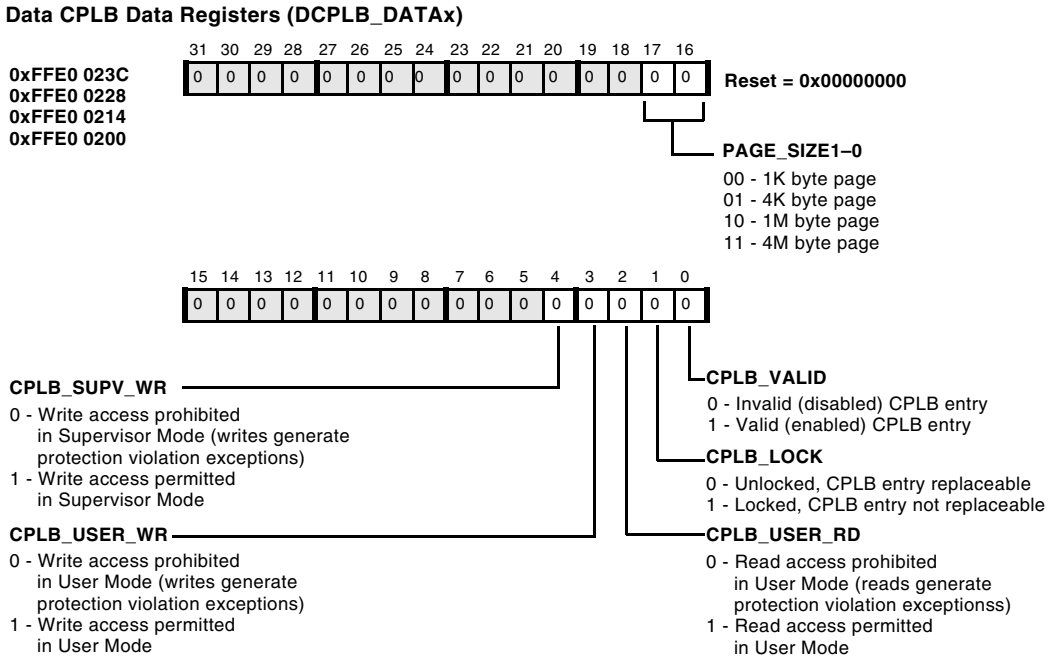

Data CPLB Data Registers (DCPLB_DATAx)
0xFFE0 023C
0xFFE0 0228
0xFFE0 0214
0xFFE0 0200
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x00000000
PAGE_SIZE1–0
00 - 1K byte page
01 - 4K byte page
10 - 1M byte page
11 - 4M byte page
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CPLB_SUPV_WR
0 - Write access prohibited in Supervisor Mode (writes generate protection violation exceptions)
1 - Write access permitted in Supervisor Mode
CPLB_USER_WR
0 - Write access prohibited in User Mode (writes generate protection violation exceptions)
1 - Write access permitted in User Mode
CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry
CPLB_LOCK
0 - Unlocked, CPLB entry replaceable
1 - Locked, CPLB entry not replaceable
CPLB_USER_RD
0 - Read access prohibited in User Mode (reads generate protection violation exceptionss)
1 - Read access permitted in User Mode

Figure 2-6. Data CPLB Data Register

ICPLB_DATAx Registers

The instruction CPLB data registers (ICPLB_DATAx), shown in Figure 2-7, contain CPLB control bits for the L1 instruction memory.

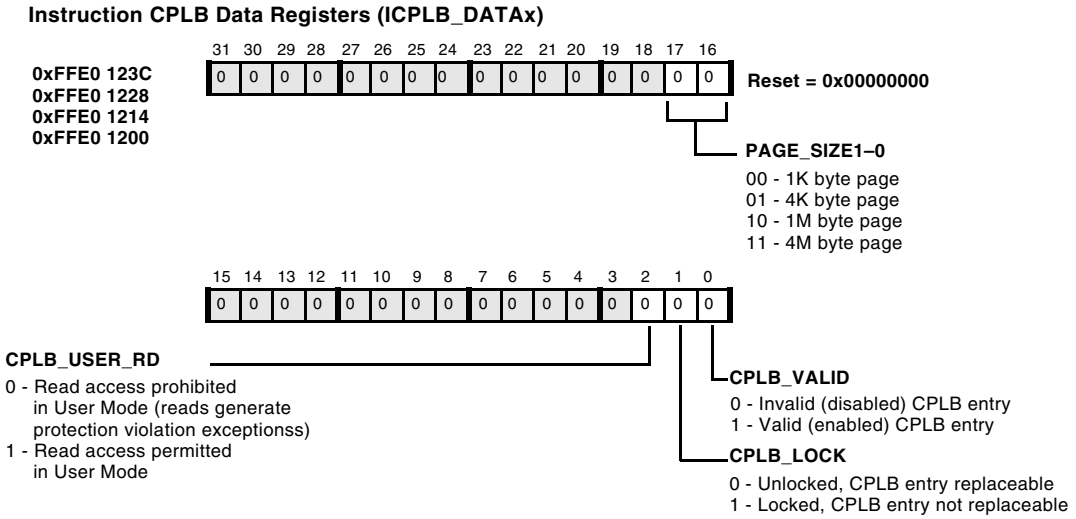

Instruction CPLB Data Registers (ICPLB_DATAx)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0xFFE0 123C
0xFFE0 1228
0xFFE0 1214
0xFFE0 1200
Reset = 0x00000000
PAGE_SIZE1–0
00 - 1K byte page
01 - 4K byte page
10 - 1M byte page
11 - 4M byte page
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CPLB_USER_RD
0 - Read access prohibited
in User Mode (reads generate
protection violation exceptionss)
1 - Read access permitted
in User Mode
CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry
CPLB_LOCK
0 - Unlocked, CPLB entry replaceable
1 - Locked, CPLB entry not replaceable

Figure 2-7. Instruction CPLB Data Register

3 CHIP BUS HIERARCHY

This chapter discusses on-chip buses, how data moves through the system, and other factors that determine the system organization. Following an overview and a list of key features is a block diagram of the chip bus hierarchy and a description of its operation. The chapter concludes with details about the system interconnects and associated system buses.

This chapter provides

- "Chip Bus Hierarchy Overview" on page 3-1
- "Interface Overview" on page 3-2

Chip Bus Hierarchy Overview

ADSP-BF59x Blackfin processors feature a powerful chip bus hierarchy on which all data movement between the processor core, internal memory, and its rich set of peripherals occurs. The chip bus hierarchy includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock domain conversion is provided to support clock domain transactions between the core and the system.

The processor system includes:

- The peripheral set including timers, TWI, UART, SPORTs, SPIs, PPI, and watchdog timer
- The Direct Memory Access (DMA) controller
- The interfaces between these and the system

The following sections describe the on-chip interfaces between the system and the peripherals via the:

- Peripheral Access Bus (PAB)
- DMA Access Bus (DAB)
- DMA Core Bus (DCB)

Interface Overview

Figure 3-1 shows the core processor and system boundaries as well as the interfaces between them.

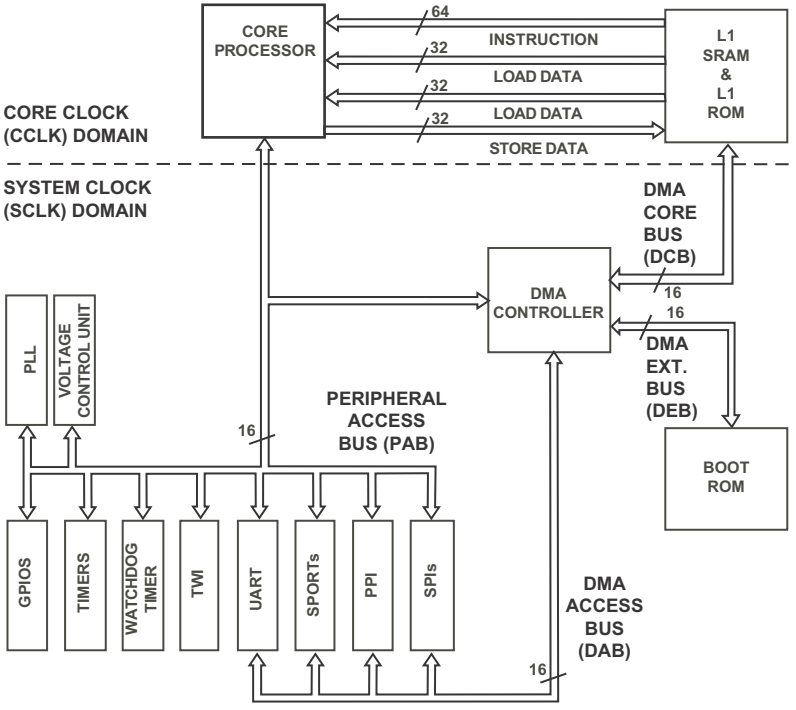


Figure 3-1. Processor Bus Hierarchy

Internal Clocks

The core processor clock (CCLK) rate is highly programmable with respect to CLKIN. The CCLK rate is divided down from the Phase Locked Loop (PLL) output rate. This divider ratio is set using the CSEL parameter of the PLL divide register.

The PAB, the DAB, and the DCB run at system clock frequency (SCLK domain). This divider ratio is set using the SSEL parameter of the PLL divide (PLL_DIV) register and must be set so that these buses run as specified in the processor data sheet, and slower than or equal to the core clock frequency.

These buses can also be cycled at a programmable frequency to reduce power consumption, or to allow the core processor to run at an optimal frequency. Note all synchronous peripherals derive their timing from the SCLK. For example, the UART clock rate is determined by further dividing this clock frequency.

Core Bus Overview

For the purposes of this discussion, level 1 memories (L1) are included in the description of the core; they have full bandwidth access from the processor core with a 64-bit instruction bus and two 32-bit data buses.

Figure 3-2 shows the core processor and its interfaces to the peripherals and external memory resources.

The core can generate up to three simultaneous off-core accesses per cycle.

The core bus structure between the processor and L1 memory runs at the full core frequency and has data paths up to 64 bits.

When the instruction request is filled, the 64-bit read can contain a single 64-bit instruction or any combination of 16-, 32-, or 64-bit (partial) instructions.

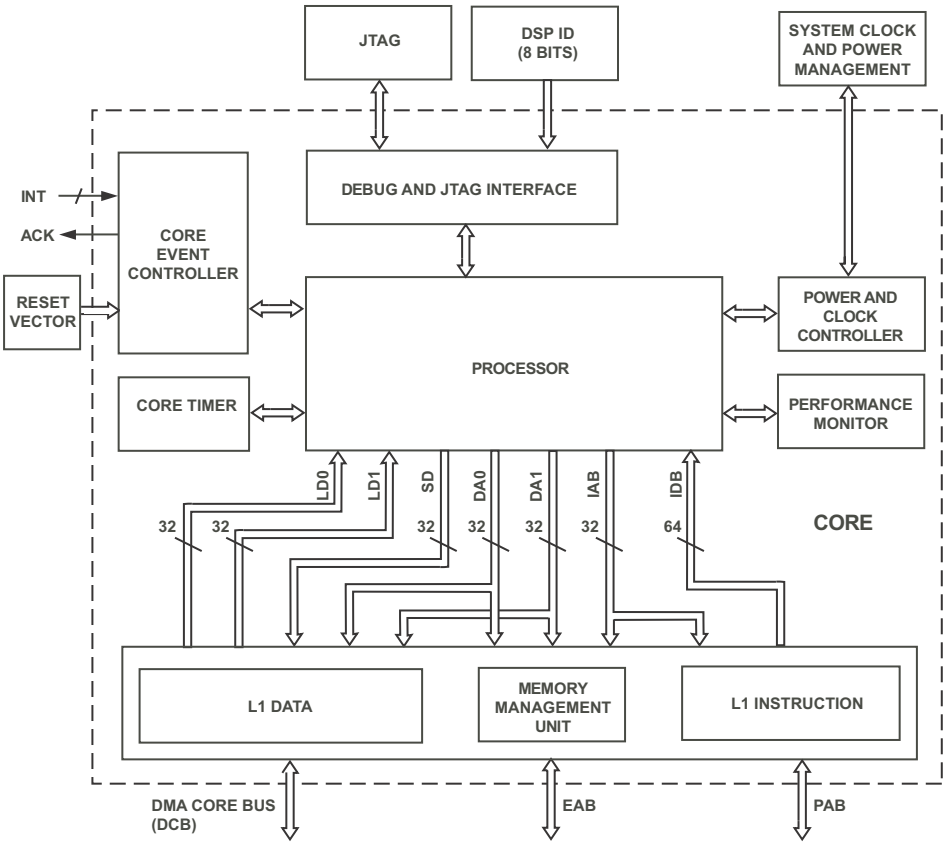


Figure 3-2. Core Block Diagram

Peripheral Access Bus (PAB)

The processor has a dedicated low latency peripheral bus that keeps core stalls to a minimum and allows for manageable interrupt latencies to time-critical peripherals. All peripheral resources accessed through the PAB are mapped into the system MMR space of the processor memory map. The core accesses system MMR space through the PAB bus.

The core processor has byte addressability, but the programming model is restricted to only 32-bit (aligned) access to the system MMRs. Byte accesses to this region are not supported.

PAB Arbitration

The core is the only master on this bus. No arbitration is necessary.

PAB Agents (Masters, Slaves)

The processor core can master bus operations on the PAB. All peripherals have a peripheral bus slave interface which allows the core to access control and status state. These registers are mapped into the system MMR space of the memory map. Appendix B lists system MMR addresses.

The slaves on the PAB bus are:

- System event controller
- Clock and power management controller
- Watchdog timer
- Timer 0–2
- SPORT0–1
- SPI0–1
- General-purpose ports
- UART
- PPI
- TWI
- DMA controller

PAB Performance

For the PAB, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the PAB are two SCLK cycles.

For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at 2x the frequency of the system clock, the first and subsequent system MMR read or write accesses take four core clocks (CCLK) of latency.

The PAB has a maximum frequency of SCLK.

DMA Access Bus (DAB), DMA Core Bus (DCB)

The DAB and DCB buses provide a means for DMA-capable peripherals to gain access to on-chip memory with little or no degradation in core bandwidth to memory.

DAB and DCB Arbitration

Thirteen DMA channels and bus masters support the DMA-capable peripherals in the processor system. The nine peripheral DMA channel controllers can transfer data between peripherals and internal memory. Both the read and write channels of the dual-stream memory DMA controller access their descriptor lists through the DAB.

The DCB has priority over the core processor on arbitration into L1 SRAM. The processor has a programmable priority arbitration policy on the DAB. Table 3-1 shows the default arbitration priority.

Table 3-1. DAB and DCB Arbitration Priority

DAB, DCB Master	Default Arbitration Priority
PPI receive or transmit	0 - highest
SPORT0 receive	1

Table 3-1. DAB and DCB Arbitration Priority (Continued)

DAB, DCB Master	Default Arbitration Priority
SPORT0 transmit	2
SPORT1 receive	3
SPORT1 transmit	4
SPI0 transmit/receive	5
SPI1 transmit/receive	6
UART0 receive	7
UART0 transmit	8
Not available on this product	9
Not available on this product	10
Not available on this product	11
Mem DMA has no peripheral mapping.	12
Mem DMA has no peripheral mapping.	13
Mem DMA has no peripheral mapping.	14
Mem DMA has no peripheral mapping.	15 - lowest

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 3-1. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

When two or more DMA master channels are actively requesting the DAB, bus utilization is considerably higher due to the DAB's pipelined design. Bus arbitration cycles are concurrent with the previous DMA access's data cycles.

DAB and DCB Performance

The processor DAB supports data transfers of 16 bits or 32 bits. The data bus has a 16-bit width with a maximum frequency as specified in the processor data sheet.

The DAB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4K byte size for L1). If there is a conflict, DMA is the highest priority requester, followed by the core.

Note that a locked transfer by the core processor effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers.

DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel. Latencies caused by these stalls are in addition to any arbitration latencies.

4 SYSTEM INTERRUPTS

This chapter discusses the system interrupt controller (SIC). While this chapter does refer to features of the core event controller (CEC), it does not cover all aspects of it. Please refer to the *Blackfin Processor Programming Reference* for more information on the CEC.

Specific Information for the ADSP-BF59x

For details regarding the number of system interrupts for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

To determine how each of the system interrupts is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each DMA, refer to Chapter A, "System MMR Assignments".

System interrupt behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 4-15.

Overview

The processor system has numerous peripherals, which therefore require many supporting interrupts.

Features

The Blackfin architecture provides a two-level interrupt processing scheme:

- The core event controller (CEC) runs in the CCLK clock domain. It interacts closely with the program sequencer and manages the event vector table (EVT). The CEC processes not only core-related interrupts such as exceptions, core errors, and emulation events; it also supports software interrupts.
- The system interrupt controller (SIC) runs in the SCLK clock domain. It masks, groups, and prioritizes interrupt requests signalled by on-chip or off-chip peripherals and forwards them to the CEC.

Description of Operation

The following sections describe the operation of the system interrupts.

Events and Sequencing

The processor employs a two-level event control mechanism. The processor SIC works with the CEC to prioritize and control all system interrupts. The SIC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. This mapping is programmable, and individual interrupt sources can be masked in the SIC.

The CEC of the processor manages five types of activities or events:

- Emulation
- Reset
- Nonmaskable interrupts (NMI)

- Exceptions
- Interrupts

Note the word *event* describes all five types of activities. The CEC manages fifteen different events in all: emulation, reset, NMI, exception, and eleven interrupts.

An interrupt is an event that changes the normal processor instruction flow and is asynchronous to program flow. In contrast, an exception is a software initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be pre-empted by one of higher priority.

The CEC supports nine general-purpose interrupts (`IVG7` – `IVG15`) in addition to the dedicated interrupt and exception events that are described in Table 4-1. It is common for applications to reserve the lowest or the two lowest priority interrupts (`IVG14` and `IVG15`) for software interrupts, leaving eight or seven prioritized interrupt inputs (`IVG7` – `IVG13`) for peripheral purposes. Refer to Table 4-1.

Table 4-1. System and Core Event Mapping

Event Source	Core Event Name
Core events	
Emulation (highest priority)	EMU
Reset	RST
NMI	NMI
Exception	EVX
Reserved	–
Hardware error	IVHW
Core timer	IVTMR

Table 4-1. System and Core Event Mapping (Continued)

Event Source	Core Event Name
System interrupts	IVG7–IVG13
Software interrupt 1	IVG14
Software interrupt 2 (lowest priority)	IVG15

System Peripheral Interrupts

To service the rich set of peripherals, the SIC has multiple interrupt request inputs and outputs that go to the CEC. The primary function of the SIC is to mask, group, and prioritize interrupt requests and to forward them to the nine general-purpose interrupt inputs of the CEC (`IVG7`–`IVG15`). Additionally, the SIC controller can enable individual peripheral interrupts to wake up the processor from Idle or power-down state.

The nine general-purpose interrupt inputs (`IVG7`–`IVG15`) of the core event controller have fixed priority. Of this group, the `IVG7` channel has the highest priority and `IVG15` has the lowest priority. Therefore, the interrupt assignment in the `SIC_IAR` registers not only groups peripheral interrupts; it also programs their priority by assigning them to individual IVG channels. However, the relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the `SIC_IAR` register settings shown in Figure 4-2 on page 4-11 and the tables in Chapter A, “System MMR Assignments”. If more than one interrupt source is mapped to the same interrupt, they are logically OR’ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

For general-purpose interrupts with multiple peripheral interrupts assigned to them, take special care to ensure that software correctly processes all pending interrupts sharing that input. Software is responsible for prioritizing the shared interrupts.

The core timer has a dedicated input to the CEC controller. Its interrupt is not routed through the SIC controller and always has higher priority than requests from all peripherals.

The SIC_IMASK register allows software to mask any peripheral interrupt source at the SIC level. This functionality is independent of whether the particular interrupt is enabled at the peripheral itself. At reset, the contents of the SIC_IMASK register are all 0s to mask off all peripheral interrupts. Turning off a system interrupt mask and enabling the particular interrupt is performed by writing a 1 to a bit location in the SIC_IMASK register.

The SIC includes one or more read-only SIC_ISR registers with individual bits which correspond to the interrupt status of one of the peripheral interrupt sources. When the SIC detects the interrupt, the bit is asserted. When the SIC detects that the peripheral interrupt input has been deasserted, the respective bit in the system interrupt status register is cleared. Note for some peripherals, such as general-purpose I/O asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt has been deasserted.

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt. One of the first instructions executed in an interrupt service routine should read the SIC_ISR register to determine whether more than one of the peripherals sharing the input has asserted its interrupt output. The service routine should fully process all pending, shared interrupts before executing the RTI, which enables further interrupt generation on that interrupt input.

When an interrupt's service routine is finished, the RTI instruction clears the appropriate bit in the IPEND register. However, the relevant SIC_ISR bit is not cleared unless the service routine clears the mechanism that generated the interrupt.

Many systems need relatively few interrupt-enabled peripherals, allowing each peripheral to map to a unique core priority level. In these designs, the SIC_ISR register will seldom, if ever, need to be interrogated.

The SIC_ISR register is not affected by the state of the SIC_IMASK register and can be read at any time. Writes to the SIC_ISR register have no effect on its contents.

Peripheral DMA channels are mapped in a fixed manner to the peripheral interrupt IDs. However, the assignment between peripherals and DMA channels is freely programmable with the DMA_PERIPHERAL_MAP registers. Table 4-1 on page 4-3 and Table 4-2 on page 4-11 show the default DMA assignment. Once a peripheral has been assigned to any other DMA channel it uses the new DMA channel's interrupt ID regardless of whether DMA is enabled or not. Therefore, clean DMA_PERIPHERAL_MAP management is required even if the DMA is not used. The default setup should be the best choice for all non-DMA applications.

For dynamic power management, any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the SIC_IWR register (refer to Table 4-1 on page 4-3 and Table 4-2 on page 4-11). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wakeup sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see the Dynamic Power Management chapter.

The SIC_IWR register has no effect unless the core is idled. By default, all interrupts generate a wakeup request to the core. However, for some

applications it may be desirable to disable this function for some peripherals, such as for a SPORT transmit interrupt. The SIC_IWR register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written to only when all peripheral interrupts are disabled.

The wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in the SIC_IWR but masked off in the SIC_IMASK register, the core wakes up if it is idled, but it does not generate an interrupt.

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 4-2 on page 4-11.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory mapped registers (MMRs) to determine which peripheral generated the interrupt.

Programming Model

The programming model for the system interrupts is described in the following sections.

System Interrupt Initialization

If the default peripheral-to-IVG assignments shown in Table 4-1 on page 4-3 and Table 4-2 on page 4-11 are acceptable, then interrupt initialization involves only:

- Initialization of the core event vector table (EVT) vector address entries

- Initialization of the IMASK register
- Unmasking the specific peripheral interrupts that the system requires in the SIC_IMASK register

System Interrupt Processing Summary

Referring to Figure 4-1 on page 4-9, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).
2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.
3. SIC_IMASK masks off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.
4. The SIC_IAR registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 - IVG15), determine the core priority of interrupt A.
5. ILAT adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to interrupt A is not masked, the process proceeds to Step 7.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's interrupt service routine (ISR).

8. When the event vector for interrupt A has entered the core pipeline, the appropriate IPEND bit is set, which clears the respective ILAT bit. Thus, IPEND tracks all pending interrupts, as well as those being presently serviced.

9. When the interrupt service routine (ISR) for interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

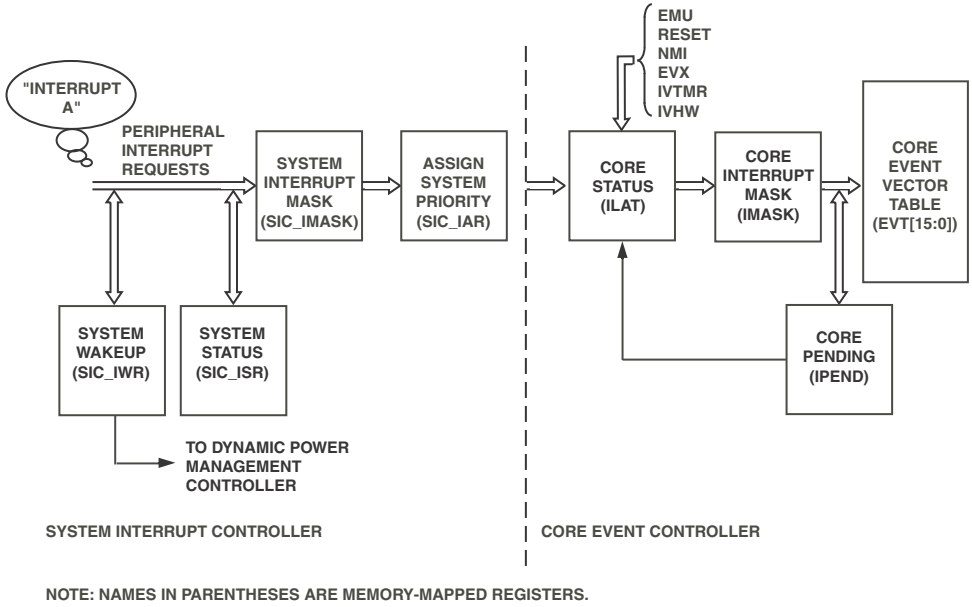


Figure 4-1. Interrupt Processing Block Diagram

It should be noted that emulation, reset, NMI, and exception events, as well as hardware error (IVHW) and core timer (IVTMR) interrupt requests, enter the interrupt processing chain at the ILAT level and are not affected

by the system-level interrupt registers (SIC_IWR, SIC_ISR, SIC_IMASK, SIC_IAR).

If multiple interrupt sources share a single core interrupt, then the interrupt service routine (ISR) must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.

System Interrupt Controller Registers

The SIC registers are described in the following sections.

These registers can be read from or written to at any time in supervisor mode. It is advisable, however, to configure them in the reset interrupt service routine before enabling interrupts. To prevent spurious or lost interrupt activity, these registers should be written to only when all peripheral interrupts are disabled.

System Interrupt Assignment (SIC_IAR) Register

The SIC_IAR register maps each peripheral interrupt ID to a corresponding IVG priority level. This is accomplished with 4-bit groupings that translate to IVG levels as shown in Table 4-2 and Figure 4-2 on page 4-11. In other words, Table 4-2 defines the value to write in a 4-bit field within SIC_IAR in order to configure a peripheral interrupt ID for a particular IVG priority. Refer to Table 4-1 on page 4-3 for information on SIC_IAR mappings for this specific processor.

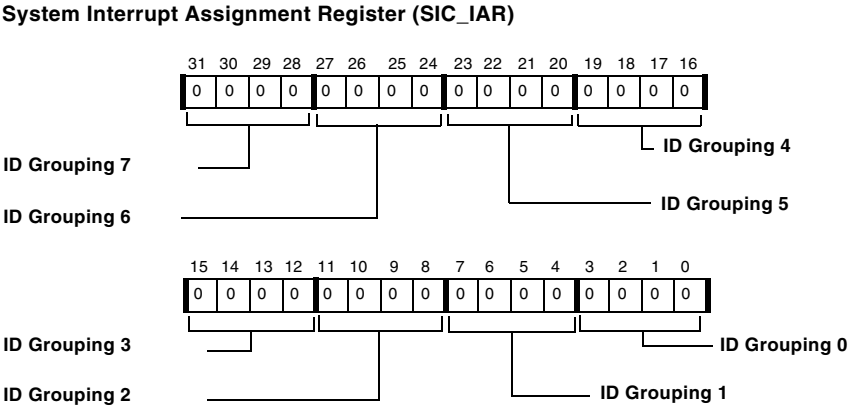

System Interrupt Assignment Register (SIC_IAR)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ID Grouping 7
ID Grouping 6
ID Grouping 5
ID Grouping 4
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ID Grouping 3
ID Grouping 2
ID Grouping 1
ID Grouping 0

Figure 4-2. System Interrupt Assignment Register

Table 4-2. IVG Select Definitions

General-purpose Interrupt	Value in SIC_IAR
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5
IVG13	6
IVG14	7
IVG15	8

System Interrupt Mask (SIC_IMASK) Register

The SIC_IMASK register masks or enables peripheral interrupts at the system level. A "0" in a bit position masks off (disables) interrupts for that particular peripheral interrupt ID. A "1" enables interrupts for that interrupt ID. Refer to Table 4-1 on page 4-3 and Table 4-2 on page 4-11 for information on how peripheral interrupt IDs are mapped to the SIC_IMASK register(s) for this particular processor.

System Interrupt Status (SIC_ISR) Register

The SIC_ISR register keeps track of system interrupts that are asserted but not yet serviced. A "0" in a bit position indicates that a particular interrupt is deasserted. A "1" indicates that it is asserted. Refer to Table 4-1 on page 4-3 and Table 4-2 on page 4-11 for information on how peripheral interrupt IDs are mapped to the SIC_ISR register(s) for this particular processor.

System Interrupt Wakeup-Enable (SIC_IWR) Register

The SIC_IWR register allows an interrupt request to wake up the processor core from an idled state. A "0" in a bit position indicates that a particular peripheral interrupt ID is not configured to wake the core (upon assertion of the interrupt request). A "1" indicates that it is configured to do so. Refer to Table 4-1 on page 4-3 and Table 4-2 on page 4-11 for information on how peripheral interrupt IDs are mapped to the SIC_IWR register(s) for this particular processor.

Programming Examples

The following section provides an example for servicing interrupt requests.

Clearing Interrupt Requests

When the processor services a core event it automatically clears the requesting bit in the ILAT register and no further action is required by the interrupt service routine. It is important to understand that the SIC controller does not provide any interrupt acknowledgment feedback mechanism from the CEC controller back to the peripherals. Although the ILAT bits clear in the same way when a peripheral interrupt is serviced, the signalling peripheral does not release its level-sensitive request until it is explicitly instructed by software. If however, the peripheral keeps requesting, the respective ILAT bit is set again immediately and the service routine is invoked again as soon as its first run terminates by an RTI instruction.

Every software routine that services peripheral interrupts must clear the signalling interrupt request in the respective peripheral. The individual peripherals provide customized mechanisms for how to clear interrupt requests. Receive interrupts, for example, are cleared when received data is read from the respective buffers. Transmit requests typically clear when software (or DMA) writes new data into the transmit buffers. These implicit acknowledge mechanisms avoid the need for cycle-consuming software handshakes in streaming interfaces. Other peripherals such as timers, GPIOs, and error requests require explicit acknowledge instructions, which are typically performed by efficient W1C (write-1-to-clear) operations.

Listing 4-1 shows a representative example of how a GPIO interrupt request might be serviced.

Listing 4-1. Servicing GPIO Interrupt Request

```
#include <defBF527.h>
/*ADSP-BF527 product is used as an example*/
.section program;
_portg_a_isr:
  /* push used registers */
  [--sp] = (r7:7, p5:5);
  /* clear interrupt request on GPIO pin PG2 */
  /* no matter whether used A or B channel */
  p5.l = lo(PORTGIO_CLEAR);
  p5.h = hi(PORTGIO_CLEAR);
  r7 = PG2;
  w[p5] = r7;

  /*  place user code here */

  /* sync system, pop registers and exit */
  ssync;
  (r7:7, p5:5) = [sp++];
  rti;
_portg_a_isr.end:
```

The W1C instruction shown in this example may require several SCLK cycles to complete, depending on system load and instruction history. The program sequencer does not wait until the instruction completes and continues program execution immediately. The SSYNC instruction ensures that the W1C command indeed cleared the request in the GPIO peripheral before the RTI instruction executes. However, the SSYNC instruction does not guarantee that the release of interrupt request has also been recognized by the CEC controller, which may require a few more CCLK cycles depending on the CCLK-to-SCLK ratio. In service routines consisting of a few instructions only, two SSYNC instructions are recommended between the clear command and the RTI instruction. However, one SSYNC instruction is typically sufficient if the clear command performs in the very beginning

of the service routine, or the SSYNC instruction is followed by another set of instructions before the service routine returns. Commonly, a pop-multiple instruction is used for this purpose as shown in Listing 4-1.

The level-sensitive nature of peripheral interrupts enables more than one of them to share the same IVG channel and therefore the same interrupt priority. This is programmable using the assignment registers. Then a common service routine typically interrogates the SIC_ISR register to determine the signalling interrupt source. If multiple peripherals are requesting interrupts at the same time, it is up to the service routine to either service all requests in a single pass or to service them one by one. If only one request is serviced and the respective request is cleared by software before the RTI instruction executes, the same service routine is invoked another time because the second request is still pending. While the first approach may require fewer cycles to service both requests, the second approach enables higher priority requests to be serviced more quickly in a non-nested interrupt system setup.

Unique Information for the ADSP-BF59x Processor

This section describes Interfaces and System Peripheral Interrupts that are unique to the ADSP-BF59x processor.

Interfaces

Figure 4-3 provides an overview of how the individual peripheral interrupt request lines connect to the SIC. It also shows how the eight SIC_IAR registers control the assignment to the nine available peripheral request inputs of the CEC.

The memory-mapped ILAT, IMASK, and IPEND registers are part of the CEC controller.

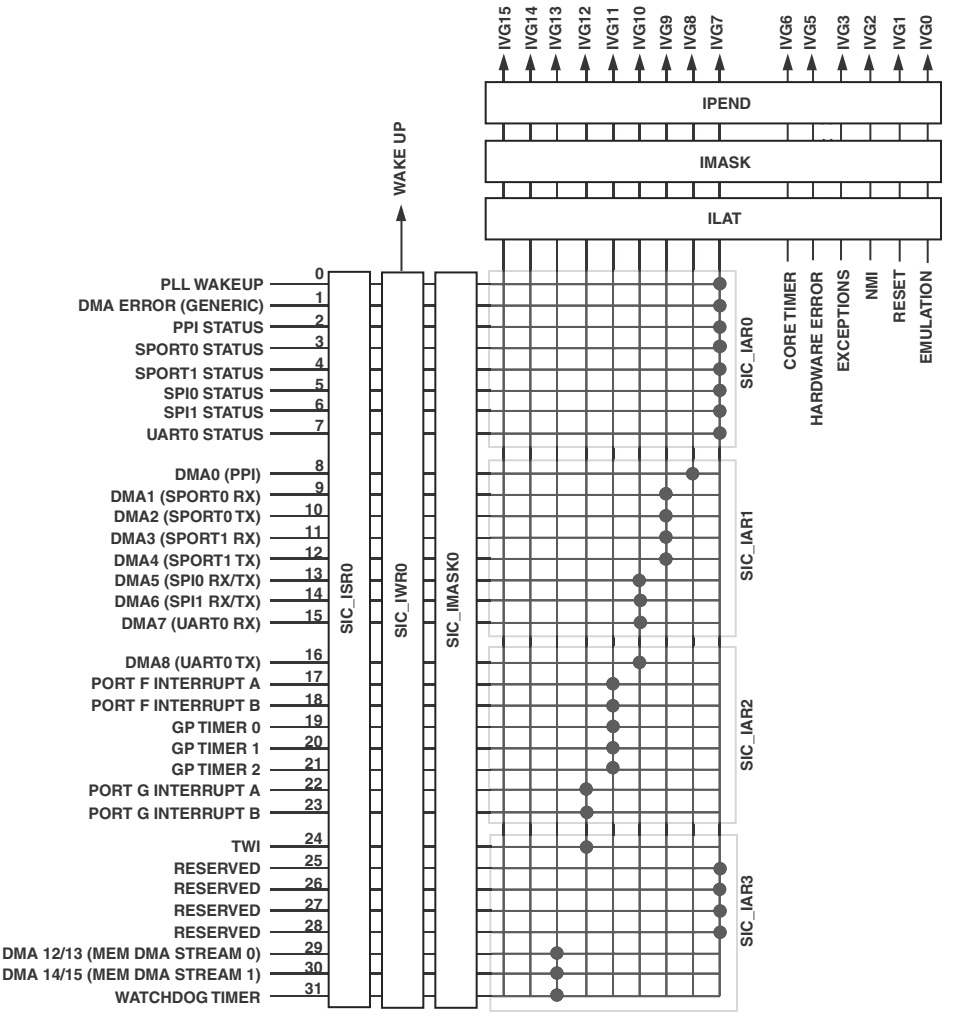


Figure 4-3. Interrupt Routing Overview

System Peripheral Interrupts

Table 4-3 shows the peripheral interrupt events, the default mapping of each event, the peripheral interrupt ID used in the system interrupt assignment registers (SIC_IAR), and the core interrupt ID.

Note that the system interrupt to core event mappings shown are the default values at reset and can be changed by software.

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in Table 4-3.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory mapped registers (MMRs) to determine which peripheral generated the interrupt.

Table 4-3. Peripheral Interrupt Events

Peripheral ID Number	Bit Position for SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3-0	Interrupt Source	Default Mapping
31	Bit 31	SIC_IAR3[31:28]	Watchdog Timer	IVG13
30	Bit 30	SIC_IAR3[27:24]	DMA 14/15 (Mem DMA Stream 1)	IVG13
29	Bit 29	SIC_IAR3[23:20]	DMA 12/13 (Mem DMA Stream 0)	IVG13
28	Bit 28	SIC_IAR3[19:16]	Reserved	IVG7
27	Bit 27	SIC_IAR3[15:12]	Reserved	IVG7
26	Bit 26	SIC_IAR3[11:8]	Reserved	IVG7
25	Bit 25	SIC_IAR3[7:4]	Reserved	IVG7
24	Bit 24	SIC_IAR3[3:0]	TWI	IVG12
23	Bit 23	SIC_IAR2[31:28]	Port G Interupt B	IVG12
22	Bit 22	SIC_IAR2[27:24]	Port G Interrupt A	IVG12
21	Bit 21	SIC_IAR2[23:20]	GP Timer 2	IVG11

Table 4-3. Peripheral Interrupt Events (Continued)

Peripheral ID Number	Bit Position for SIC_ISR0, SIC_IMASK0, SIC_IWR0	SIC_IAR3-0	Interrupt Source	Default Mapping
20	Bit 20	SIC_IAR2[19:16]	GP Timer 1	IVG11
19	Bit 19	SIC_IAR2[15:12]	GP Timer 0	IVG11
18	Bit 18	SIC_IAR2[11:8]	Port F Interrupt B	IVG11
17	Bit 17	SIC_IAR2[7:4]	Port F Interrupt A	IVG11
16	Bit 16	SIC_IAR2[3:0]	DMA8 (UART0 TX)	IVG10
15	Bit 15	SIC_IAR1[31:28]	DMA7 (UART0 RX)	IVG10
14	Bit 14	SIC_IAR1[27:24]	DMA6 (SPI1 RX/TX)	IVG10
13	Bit 13	SIC_IAR1[23:20]	DMA5 (SPI0 RX/TX)	IVG10
12	Bit 12	SIC_IAR1[19:16]	DMA4 (SPORT1 TX)	IVG9
11	Bit 11	SIC_IAR1[15:12]	DMA3 (SPORT1 RX)	IVG9
10	Bit 10	SIC_IAR1[11:8]	DMA2 (SPORT0 TX)	IVG9
9	Bit 9	SIC_IAR1[7:4]	DMA1 (SPORT0 RX)	IVG9
8	Bit 8	SIC_IAR1[3:0]	DMA0 (PPI)	IVG8
7	Bit 7	SIC_IAR0[31:28]	UART0 Status	IVG7
6	Bit 6	SIC_IAR0[27:24]	SPI1 Status	IVG7
5	Bit 5	SIC_IAR0[23:20]	SPI0 Status	IVG7
4	Bit 4	SIC_IAR0[19:16]	SPORT1 Status	IVG7
3	Bit 3	SIC_IAR0[15:12]	SPORT0 Status	IVG7
2	Bit 2	SIC_IAR0[11:8]	PPI Status	IVG7
1	Bit 1	SIC_IAR0[7:4]	DMA Error (generic)	IVG7
0	Bit 0	SIC_IAR0[3:0]	PLL Wakeup	IVG7

5 DIRECT MEMORY ACCESS

This chapter describes the direct memory access (DMA) controller. Following an overview and list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in Chapter 3, "Chip Bus Hierarchy".

Specific Information for the ADSP-BF59x

For details regarding the number of DMA controllers for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For DMA interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

To determine how each of the DMAs is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each DMA, refer to Chapter A, "System MMR Assignments".

DMA controller behavior for the ADSP-BF59x that differs from the general information in this chapter can be found in the section "Unique Information for the ADSP-BF59x Processor" on page 5-101

Overview and Features

The processor uses DMA to transfer data between memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controller carries out the data transfers independent of processor activity.

The DMA controller can perform several types of data transfers:

- Peripheral DMA transfers data between memory and on-chip peripherals.
- Memory DMA (MDMA) transfers data between memory and memory. The processor has two MDMA modules, each consisting of independent memory read and memory write channels.
- Handshaking memory DMA (HMDMA) transfers data between off-chip peripherals and memory. This enhancement of the MDMA channels enables external hardware to control the timing of individual data transfers or block transfers.

The HMDMA feature is not available for all products. Refer to "Unique Information for the ADSP-BF59x Processor" on page 5-101 to determine whether it applies to this product.

All DMAs can transport data to and from on-chip and off-chip memories, including L1 and SDRAM. The L1 scratchpad memory cannot be accessed by DMA.

SDRAM and SRAM are not available on all products. Refer to "Unique Information for the ADSP-BF59x Processor" on page 5-101 to determine whether it applies to this product.

DMA transfers on the processor can be descriptor-based or register-based.

Register-based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed.

Descriptor-based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor-based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes.

Examples of DMA styles supported by flex descriptors include:

- A single linear buffer that stops on completion (`FLOW` = stop mode)
- A linear buffer with byte strides of any integer value, including negative values (`DMAx_X_MODIFY` register)
- A circular, auto-refreshing buffer that interrupts on each full buffer
- A similar buffer that interrupts on fractional buffers (for example, ½, ¼) (2-D DMA)
- 1-D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing a link pointer and a 32-bit address
- 1-D DMA, using a linked list of 5-word descriptors containing a link pointer, a 32-bit address, the buffer length, and a configuration

- 2-D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page
- 2-D DMA, using a linked list of 9-word descriptors specifying everything

DMA Controller Overview

A block diagram of the DMA controller can be found in the "Unique Information for the ADSP-BF59x Processor" on page 5-101.

External Interfaces

The DMA does not connect external memories and devices directly. Rather, data is passed through the EBIU port. Any kind of device that is supported by the EBIU can also be accessed by peripheral DMA or memory DMA operation. This is typically flash memory, SRAM, SDRAM, FIFOs, or memory-mapped peripheral devices.

For products with handshaking MDMA (HMDMA), the operation is supported by two MDMA request input pins, `DMAR0` and `DMAR1`. The `DMAR0` pin controls transfer timing on the `MDMA0` destination channel. The `DMAR1` pin controls the destination channel of `MDMA1`. With these pins, external FIFO devices, ADC or DAC converters, or other streaming or block-processing devices can use the MDMA channels to exchange their data or data buffers with the Blackfin processor memory.

Internal Interfaces

Figure 3-1 on page 3-2 shows the dedicated DMA buses used by the DMA controller to interconnect L1 memory, the on-chip peripherals, and the EBIU port.

The 16-bit DMA core bus (DCB) connects the DMA controller to a dedicated port of L1 memory. L1 memory has dedicated DMA ports featuring special DMA buffers to decouple DMA operation. See the *Blackfin Processor Programming Reference* for a description of the L1 memory architecture. The DCB bus operates at core clock (CCLK) frequency. It is the DMA controller's responsibility to translate DCB transfers to the system clock (SCLK) domain.

The 16-bit DMA access bus (DAB) connects the DMA controller to the on-chip peripherals. This bus operates at SCLK frequency.

The 16-bit DMA external bus (DEB) connects the DMA controller to the EBIU port. This bus is used for all peripheral and memory DMA transfers to and from external memories and devices. It operates at SCLK frequency.

Transferred data can be 8-, 16-, or 32-bits wide. The DMA controller, however, connects only to 16-bit buses.

Memory DMA can pass data every SCLK cycle between L1 memory and the EBIU. Transfers from L1 memory to L1 memory require two cycles, as the DCB bus is used for both source and destination transfers. Similarly, transfers between two off-chip devices require EBIU and DEB resources twice. Peripheral DMA transfers can be performed every other SCLK cycle.

For more details on DMA performance see "DMA Performance" on page 5-40.

Peripheral DMA

The DMA controller features 12 channels that perform transfers between peripherals and on-chip or off-chip memories. The user has full control over the mapping of DMA channels and peripherals. The default DMA channel priority and mapping, shown in Table 5-9 on page 5-103, can be changed by altering the 4-bit PMAP field in the DMAx_PERIPHERAL_MAP registers for the peripheral DMA channels.

The default configuration should suffice in most cases, but there are some cases where remapping the assignment can be helpful because of the DMA channel priorities. When competing for any of the system buses, DMA0 has higher priority than DMA1, and so on. DMA11 has the lowest priority of the peripheral DMA channels.

A 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If multiple channels are mapped to the same peripheral, only one channel is connected (the lowest priority channel). If a nonexistent peripheral (for example, 0xF in the PMAP field) is mapped to a channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.

All peripheral DMA channels work completely independently from each other. The transfer timing is controlled by the mapped peripheral.

Every DMA channel features its own 4-deep FIFO that decouples DAB activity from DCB and DEB availability. DMA interrupt and descriptor fetch timing is aligned with the memory side (DCB/DEB side) of the FIFO. The user does, however, have an option to align interrupts with the peripheral side (DAB side) of the FIFO for transmit operations. Refer to the SYNC bit in the DMAx_CONFIG register for details.

Memory DMA

This section describes the two pairs of MDMA channels, which provide memory-to-memory DMA transfers among the various memory spaces. These include L1 memory and external synchronous/asynchronous memories.

Each MDMA channel contains a DMA FIFO, an 8-word by 16-bit FIFO block used to transfer data to and from either L1 or the DCB and DEB

buses. Typically, it is used to transfer data between external memory and internal memory. It will also support DMA from the boot ROM on the DEB bus. The FIFO can be used to hold DMA data transferred between two L1 memory locations or between two external memory locations.

Each page of MDMA channels consists of:

- A source channel (for reading from memory)
- A destination channel (for writing to memory)

A memory-to-memory transfer always requires both the source and the destination channel to be enabled. Each source/destination channel forms a "stream," and these two streams are hardwired for DMA priorities 12 through 15.

- Priority 12: MDMA0 destination
- Priority 13: MDMA0 source
- Priority 14: MDMA1 destination
- Priority 15: MDMA1 source

MDMA0 takes precedence over MDMA1, unless round-robin scheduling is used or priorities become urgent, as programmed by the `DRQ` bit field in the `HMDMA_CONTROL` register.

It is illegal to program a source channel for memory write or a destination channel for memory read.

The channels support 8-, 16-, and 32-bit memory DMA transfers, but both ends of the MDMA connect to 16-bit buses. Source and destination channels must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of the MDMA FIFO for a given stream are granted priority at the same time. Each pair shares an 8-word deep 16-bit FIFO. The source DMA engine fills the FIFO, while the

destination DMA engine empties it. The FIFO depth allows the burst transfers of the external access bus (EAB) and DMA access bus (DAB) to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total transfer count is the same.

To start a MDMA transfer operation, the MMRs for the source and destination channels are written, each in a manner similar to peripheral DMA.

The DMAx_CONFIG register for the source channel must be written before the DMAx_CONFIG register for the destination channel.

Handshaked Memory DMA (HMDMA) Mode

This feature is not available for all products. Refer to the "Unique Information for the ADSP-BF59x Processor" on page 5-101 to determine whether it applies to this product.

Handshaked operation applies only to memory DMA channels.

Normally, memory DMA transfers are performed at maximum speed. Once started, data is transferred in a continuous manner until either the data count expires or the MDMA is stopped. In handshake mode, the MDMA does not transfer data automatically when enabled; it waits for an external trigger on the MDMA request input signals. The DMAR0 input is associated with MDMA0 and the DMAR1 input with MDMA1. Once a trigger event is detected, a programmable portion of data is transferred and then the MDMA stalls again and waits for the next trigger.

Handshake operation is not only useful for controlling the timing of memory-to-memory transfers, it also enables the MDMA to operate with

asynchronous FIFO-style devices connected to the EBIU port. The Blackfin processor acknowledges a DMA request by a proper number of read or write operations. It is up to the device connected to any of the AMSx strobes to deassert or pulse the request signal and to decrement the number of pending requests accordingly.

Depending on HMDMA operating mode, an external DMA request may trigger individual data word transfers or block transfers. A block can consist of up to 65535 data words. For best throughput, DMA requests can be pipelined. The HMDMA controllers feature a request counter to decouple request timing from the data transfers.

See "Handshaked Memory DMA Operation" on page 5-36 for a functional description.

Modes of Operation

The following sections describe the DMA operation.

Register-Based DMA Operation

Register-based DMA is the traditional kind of DMA operation. Software configures the source or destination address and the length of the data to be transferred to memory-mapped registers and then starts DMA operation.

For basic operation, the software performs these steps:

- Write the source or destination address to the 32-bit DMAx_START_ADDR register.
- Write the number of data words to be transferred to the 16-bit DMAx_X_COUNT register.

- Write the address modifier to the 16-bit DMAx_X_MODIFY register. This is the two's-complement value added to the address pointer after every transfer. This value must always be initialized as there is no default value. Typically, this register is set to 0x0004 for 32-bit DMA transfers, to 0x0002 for 16-bit transfers, and to 0x0001 for byte transfers.
- Write the operation mode to the DMAx_CONFIG register. These bits in particular need to be changed as needed:
    - The DMAEN bit enables the DMA channel.
    - The WNR bit controls the DMA direction. DMAs that read from memory (peripheral transmit DMAs and source channel MDMAs) keep this bit cleared. Peripheral receive DMAs and destination channel MDMAs set this bit because they write to memory.
    - The WDSIZE bit controls the data word width for the transfer. It can be 8-, 16-, or 32-bits wide.
    - The DI_EN bit enables an interrupt when the DMA operation has finished.
    - Set the FLOW field to 0x0 for stop mode or 0x1 for autobuffer mode.

Once the DMAEN bit is set, the DMA channel starts its operation. While running, the DMAx_CURR_ADDR and the DMAx_CURR_X_COUNT registers can be monitored to determine the current progress of the DMA operation. However they should not be used to synchronize software and hardware.

The DMAx_IRQ_STATUS register signals whether the DMA has finished (DMA_DONE bit), whether a DMA error has occurred (DMA_ERR bit), and whether the DMA is currently running (DMA_RUN bit). The DMA_DONE and the DMA_ERR bits also function as interrupt latch bits. They must be cleared by write-one-to-clear (W1C) operations by the interrupt service routine.

Stop Mode

In stop mode, the DMA operation is executed only once. When started, the DMA channel transfers the desired number of data words and stops itself when the transfer is complete. If the DMA channel is no longer used, software should clear the DMAEN enable bit to disable the otherwise paused channel. Stop mode is entered if the FLOW bit field in the DMA channel's DMAx_CONFIG register is 0. The NDSIZE field must always be 0 in this mode.

For receive (memory write) operation, the DMA_RUN bit functions almost the same as the inverted DMA_DONE bit. For transmit (memory read) operation, however, the two bits have different timing. Refer to the description of the SYNC bit in the DMAx_CONFIG register for details.

Autobuffer Mode

In autobuffer mode, the DMA operates repeatedly in a circular manner. If all data words have been transferred, the address pointer DMAx_CURR_ADDR is reloaded automatically by the DMAx_START_ADDR value. An interrupt may also be generated.

Autobuffer mode is entered if the FLOW field in the DMAx_CONFIG register is 1. The NDSIZE bit must be 0 in autobuffer mode.

Two-Dimensional DMA Operation

Register-based and descriptor-based DMA can operate in one-dimensional mode or two-dimensional mode.

In two-dimensional (2-D) mode, the DMAx_X_COUNT register is accompanied by the DMAx_Y_COUNT register, supporting arbitrary row and column sizes up to 64K × 64K elements, as well as arbitrary DMAx_X_MODIFY and DMAx_Y_MODIFY values up to ±32K bytes. Furthermore, DMAx_Y_MODIFY can be negative, allowing implementation of interleaved datastreams. The DMAx_X_COUNT and DMAx_Y_COUNT values specify the row and column sizes, where DMAx_X_COUNT must be 2 or greater.

The start address and modify values are in bytes, and they must be aligned to a multiple of the DMA transfer word size (WDSIZE[1:0] in DMAx_CONFIG). Misalignment causes a DMA error.

The DMAx_X_MODIFY value is the byte-address increment that is applied after each transfer that decrements the DMAx_CURR_X_COUNT register. The DMAx_X_MODIFY value is not applied when the inner loop count is ended by decrementing DMAx_CURR_X_COUNT from 1 to 0, except that it is applied on the final transfer when DMAx_CURR_Y_COUNT is 1 and DMAx_CURR_X_COUNT decrements from 1 to 0.

The DMAx_Y_MODIFY value is the byte-address increment that is applied after each decrement of the DMAx_CURR_Y_COUNT register. However, the DMAx_Y_MODIFY value is not applied to the last item in the array on which the outer loop count (DMAx_CURR_Y_COUNT) also expires by decrementing from 1 to 0.

After the last transfer completes, DMAx_CURR_Y_COUNT = 1, DMAx_CURR_X_COUNT = 0, and DMAx_CURR_ADDR is equal to the last item's address plus DMAx_X_MODIFY.

If the DMA channel is programmed to refresh automatically (auto-buffer mode), then these registers will be loaded from DMAx_X_COUNT, DMAx_Y_COUNT, and DMAx_START_ADDR upon the first data transfer.

The DI_SEL configuration bit enables DMA interrupt requests every time the inner loop rolls over. If DI_SEL is cleared, but DI_EN is still set, only one interrupt is generated after the outer loop completes.

Examples of Two-Dimensional DMA

Example 1: Retrieve a 16 × 8 block of bytes from a video frame buffer of size (N × M) pixels:

```
DMAx_X_MODIFY = 1
DMAx_X_COUNT = 16
```

`DMAx_Y_MODIFY = N-15` (offset from the end of one row to the start of another)
`DMAx_Y_COUNT = 8`

This produces the following address offsets from the start address:

```
0,1,2,...15,
N,N + 1, ... N + 15,
2N, 2N + 1,... 2N + 15, ...
7N, 7N + 1,... 7N + 15,
```

Example 2: Receive a video datastream of bytes,
(R,G,B pixels) × (N × M image size):

```
DMAx_X_MODIFY = (N * M)
DMAx_X_COUNT = 3
DMAx_Y_MODIFY = 1 - 2(N * M) (negative)
DMAx_Y_COUNT = (N * M)
```

This produces the following address offsets from the start address:

```
0, (N * M), 2(N * M),
1, (N * M) + 1, 2(N * M) + 1,
2, (N * M) + 2, 2(N * M) + 2,
    ...
(N * M) - 1, 2(N * M) - 1, 3(N * M) - 1,
```

Descriptor-based DMA Operation

In descriptor-based DMA operation, software does not set up DMA sequences by writing directly into DMA controller registers. Rather, software keeps DMA configurations, called descriptors, in memory. On demand, the DMA controller loads the descriptor from memory and overwrites the affected DMA registers by its own control. Descriptors can be fetched from L1 memory using the DCB bus or from external memory using the DEB bus.

A descriptor describes what kind of operation should be performed next by the DMA channel. This includes the DMA configuration word as well as data source/destination address, transfer count, and address modify values. A DMA sequence controlled by one descriptor is called a work unit.

Optionally, an interrupt can be requested at the end of any work unit by setting the DI_EN bit in the configuration word of the respective descriptor.

A DMA channel is started in descriptor-based mode by first writing the 32-bit address of the first descriptor into the DMAx_NEXT_DESC_PTR register (or the DMAx_CURR_DESC_PTR in case of descriptor array mode) and then performing a write to the DMAx_CONFIG register that sets the FLOW field to either 0x4, 0x6, or 0x7 and enables the DMAEN bit. This causes the DMA controller to immediately fetch the descriptor from the address pointed to by the DMAx_NEXT_DESC_PTR register. The fetch overwrites the DMAx_CONFIG register again. If the DMAEN bit is still set, the channel starts DMA processing.

The DFETCH bit in the DMAx_IRQ_STATUS register tells whether a descriptor fetch is ongoing on the respective DMA channel. The DMAx_CURR_DESC_PTR points to the descriptor value that is to be fetched next.

Descriptor List Mode

Descriptor list mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to either 0x6 (small descriptor mode) or 0x7 (large descriptor mode). In either of these modes multiple descriptors form a chained list. Every descriptor contains a pointer to the next descriptor. When the descriptor is fetched, this pointer value is loaded into the DMAx_NEXT_DESC_PTR register of the DMA channel. In large descriptor mode this pointer is 32 bits wide. Therefore, the next descriptor may reside in any address space accessible through the DCB and DEB buses. In small descriptor mode this pointer is just 16 bits wide. For this reason, the next descriptor must reside in the same 64K byte address space

as the first one because the upper 16 bits of the DMAx_NEXT_DESC_PTR register are not updated.

Descriptor list modes are started by writing first to the DMAx_NEXT_DESC_PTR register and then to the DMAx_CONFIG register.

Descriptor Array Mode

Descriptor array mode is selected by setting the FLOW bit field in the DMA channel's DMAx_CONFIG register to 0x4. In this mode, the descriptors do not contain further descriptor pointers. The initial DMAx_CURR_DESC_PTR value is written by software. It points to an array of descriptors. The individual descriptors are assumed to reside next to each other and, therefore, their addresses are known.

Variable Descriptor Size

In any descriptor-based mode the NDSIZE field in the configuration word specifies how many 16-bit words of the next descriptor need to be loaded on the next fetch. In descriptor-based operation, NDSIZE must be non-zero. The descriptor size can be any value from one entry (the lower 16 bits of DMAx_START_ADDR only) to nine entries (all the DMA parameters). Table 5-1 illustrates how a descriptor must be structured in memory. The values have the same order as the corresponding MMR addresses.

If, for example, a descriptor is fetched in array mode with NDSIZE = 0x5, the DMA controller fetches the 32-bit start address, the DMA configuration word, and the XCNT and XMOD values. However, it does not load YCNT and YMOD. This might be the case if the DMA operates in one-dimensional mode or if the DMA is in two-dimensional mode, but the YCNT and YMOD values do not need to change.

All the other registers not loaded from the descriptor retain their prior values, although the DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, and

DMAx_CURR_Y_COUNT registers are reloaded between the descriptor fetch and the start of DMA operation.

Table 5-1 shows the offsets for descriptor elements in the three modes described above. Note the names in the table describe the descriptor elements in memory, not the actual MMRs into which they are eventually loaded. For more information regarding descriptor element acronyms, see Table 5-6 on page 5-63.

Table 5-1. Parameter Registers and Descriptor Offsets

Descriptor Offset	Descriptor Array Mode	Small Descriptor List Mode	Large Descriptor List Mode
0x0	SAL	NDPL	NDPL
0x2	SAH	SAL	NDPH
0x4	DMACFG	SAH	SAL
0x6	XCNT	DMACFG	SAH
0x8	XMOD	XCNT	DMACFG
0xA	YCNT	XMOD	XCNT
0xC	YMOD	YCNT	XMOD
0xE		YMOD	YCNT
0x10			YMOD

Note that every descriptor fetch consumes bandwidth from either the DCB bus or the DEB bus and the external memory interface, so it is best to keep the size of descriptors as small as possible.

Mixing Flow Modes

The FLOW mode of a DMA is not a global setting. If the DMA configuration word is reloaded with a descriptor fetch, the FLOW and NDSIZE bit fields can also be altered. A small descriptor might be used to loop back to the first descriptor if a descriptor array is used in an endless manner. If the

descriptor chain is not endless and the DMA is required to stop after a certain descriptor has been processed, the last descriptor is typically processed in stop mode. That is, its FLOW and NDSIZE fields are 0, but its DMAEN bit is still set.

Functional Description

The following sections provide a functional description of DMA.

DMA Operation Flow

Figure 5-1 and Figure 5-2 describe the DMA flow.

DMA Startup

This section discusses starting DMA "from scratch." This is similar to starting it after it has been paused by the FLOW = 0 mode.

Before initiating DMA for the first time on a given channel, all parameter registers must be initialized . Be sure to initialize the upper 16 bits of the DMAx_NEXT_DESC_PTR (or DMAx_CURR_DESC_PTR register in FLOW = 4 mode) and DMAx_START_ADDR registers, because they might not otherwise be accessed, depending upon the flow mode. Also note that the DMAx_X_MODIFY and DMAx_Y_MODIFY registers are not preset to a default value at reset.

The user may wish to write other DMA registers that might be static during DMA activity (for example, DMAx_X_MODIFY, DMAx_Y_MODIFY). The contents of NDSIZE and FLOW in DMAx_CONFIG indicate which registers, if any, are fetched from descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMAx_CONFIG with DMAEN = 1.

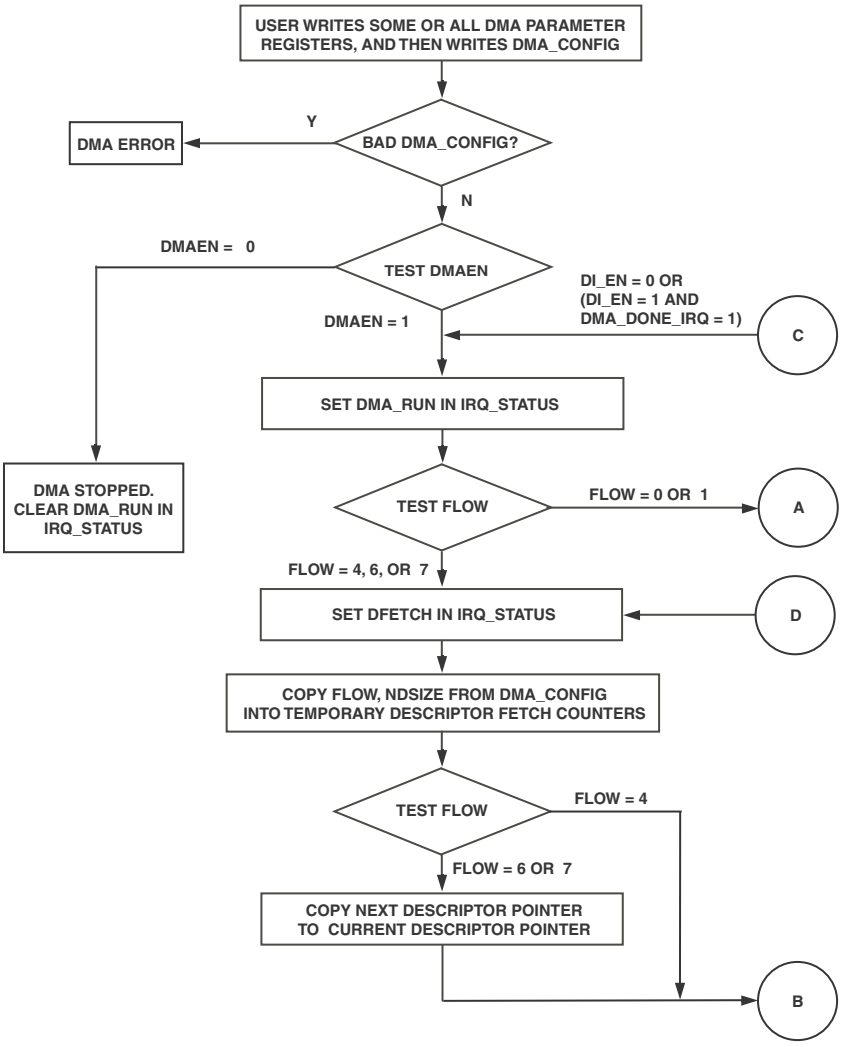

USER WRITES SOME OR ALL DMA PARAMETER REGISTERS, AND THEN WRITES DMA_CONFIG
BAD DMA_CONFIG?
Y
DMA ERROR
N
TEST DMAEN
DMAEN = 0
DMAEN = 1
DI_EN = 0 OR (DI_EN = 1 AND DMA_DONE_IRQ = 1)
C
SET DMA_RUN IN IRQ_STATUS
DMA STOPPED. CLEAR DMA_RUN IN IRQ_STATUS
TEST FLOW
FLOW = 0 OR 1
A
FLOW = 4, 6, OR 7
SET DFETCH IN IRQ_STATUS
D
COPY FLOW, NDSIZE FROM DMA_CONFIG INTO TEMPORARY DESCRIPTOR FETCH COUNTERS
TEST FLOW
FLOW = 4
FLOW = 6 OR 7
COPY NEXT DESCRIPTOR POINTER TO CURRENT DESCRIPTOR POINTER
B

Figure 5-1. DMA Flow, From DMA Controller's Point of View (1 of 2)

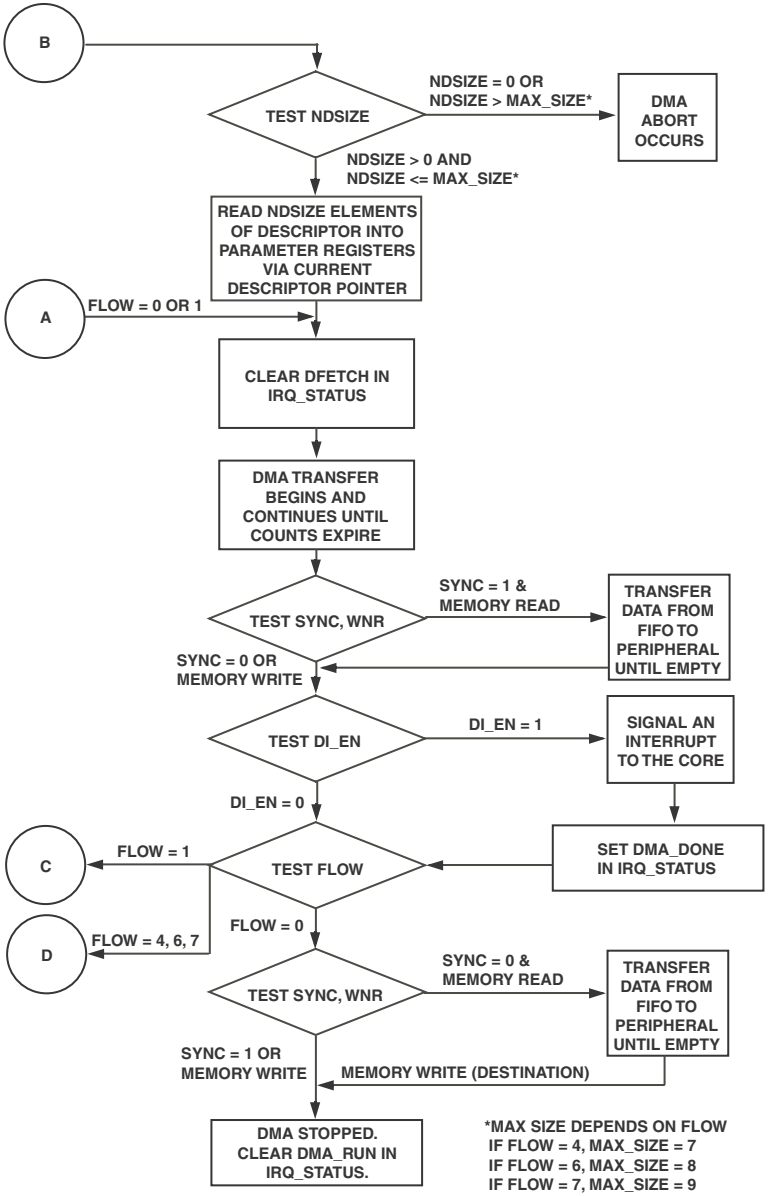


Figure 5-2. DMA Flow, From DMA Controller's Point of View (2 of 2)

When DMAx_CONFIG is written directly by software, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMAEN = 1, the DMACFG descriptor element that was read into DMAx_CONFIG assumes control. Before this point, the direct write to DMAx_CONFIG had control. In other words, the WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be taken from the DMACFG value in the descriptor read from memory, while these field values initially written to the DMAx_CONFIG register are ignored.

As Figure 5-1 on page 5-18 and Figure 5-2 on page 5-19 show, at startup the FLOW and NDSIZE bits in DMAx_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more current registers from descriptor elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values.

If the FLOW value specifies small or large descriptor list modes, the DMAx_NEXT_DESC_PTR is copied into DMAx_CURR_DESC_PTR. Then, fetches of new descriptor elements from memory are performed, indexed by DMAx_CURR_DESC_PTR, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into DMAx_NEXT_DESC_PTR, but the fetch of the current descriptor continues using DMAx_CURR_DESC_PTR. After completion of the descriptor fetch, DMAx_CURR_DESC_PTR points to the next 16-bit word in memory past the end of the descriptor.

If neither NDPH nor NDPL are part of the descriptor (that is, in descriptor array mode, FLOW = 4), then the transfer from NDPH/NDPL into DMAx_CURR_DESC_PTR does not occur. Instead, descriptor fetch indexing begins with the value in DMAx_CURR_DESC_PTR.

If DMACFG is not part of the descriptor, the previous DMAx_CONFIG settings (as written by MMR access at startup) control the work unit operation. If DMACFG is part of the descriptor, then the DMAx_CONFIG value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor's DMACFG and by the parameter registers loaded from the descriptor. The bits DI_EN, DI_SEL, DMA2D, WDSIZE, and WNR in the value programmed by the MMR access are disregarded.

The DMA_RUN and DFETCH status bits in the DMAx_IRQ_STATUS register indicate the state of the DMA channel. After a write to DMAx_CONFIG, the DMA_RUN and DFETCH bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, DMA data transfer operation begins. The DMA channel immediately attempts to fill its FIFO, subject to channel priority—a memory write (RX) DMA channel begins accepting data from its peripheral, and a memory read (TX) DMA channel begins memory reads, provided the channel wins the grant for bus access.

When the DMA channel performs its first data memory access, its address and count computations take their input operands from the start registers (DMAx_START_ADDR, DMAx_X_COUNT, DMAx_Y_COUNT), and write results back to the current registers (DMAx_CURR_ADDR, DMAx_CURR_X_COUNT, DMAx_CURR_Y_COUNT). Note also that the current registers are not valid until the first memory access is performed, which may be some time after the channel is started by the write to the DMA_CONFIG register. The current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows.

- DMAx_START_ADDR is copied to DMAx_CURR_ADDR
- DMAx_X_COUNT is copied to DMAx_CURR_X_COUNT
- DMAx_Y_COUNT is copied to DMAx_CURR_Y_COUNT

Then DMA data transfer operation begins, as shown in Figure 5-2 on page 5-19.

DMA Refresh

On completion of a work unit:

- The DMA controller completes the transfer of all data between memory and the DMA unit.
- If SYNC = 1 and WNR = 0 (memory read), the DMA controller selects a synchronized transition and transfers all data to the peripheral before continuing.
- If enabled by DI_EN, the DMA controller signals an interrupt to the core and sets the DMA_DONE bit in the channel's DMAx_IRQ_STATUS register.
- If FLOW = 0 the DMA controller stops operation by clearing the DMA_RUN bit in DMAx_IRQ_STATUS register after all data in the channel's DMA FIFO has been transferred to the peripheral.
- During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in DMAx_IRQ_STATUS register to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

  If FLOW = 4 (descriptor array) the DMA controller loads a new descriptor from memory into the DMA registers using the contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMAx_CONFIG register prior to the beginning of the fetch.

  If FLOW = 6 (small descriptor list) the DMA controller copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, the DMA controller fetches a descriptor from memory into the DMA

registers using the new contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The first descriptor element that is loaded is a new 16-bit value for the lower 16 bits of DMAx_NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of DMAx_NEXT_DESC_PTR will retain their former value. This supports a shorter, more efficient descriptor than the large descriptor list model, which is suitable whenever the application can place the channel's descriptors in the same 64K byte range of memory.

If FLOW = 7 (large descriptor list) the DMA controller copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, the DMA controller fetches a descriptor from memory into the DMA registers using the new contents of DMAx_CURR_DESC_PTR, and increments DMAx_CURR_DESC_PTR. The first descriptor element that is loaded is a new 32-bit value for the full DMAx_NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of DMAx_NEXT_DESC_PTR may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal memory or external memory.

- If it is necessary to link from a descriptor chain whose descriptors are in one 64K byte area to another chain whose descriptors are outside that area, only the descriptor containing the link to the new 64K byte range needs to use FLOW = 7. All descriptors that reference the same 64K byte area may use FLOW = 6.
- If FLOW = 4, 6, or 7 (descriptor array, small descriptor list, or large descriptor list, respectively), the DMA controller clears the DFETCH bit in the DMAx_IRQ_STATUS register.
- If FLOW = any value but 0 (Stop), the DMA controller begins the next work unit for that channel, which must contend with other channels for priority on the memory buses. On the first memory transfer of the new work unit, the DMA controller updates the

current registers from the start registers:

DMAx_CURR_ADDR loaded from DMAx_START_ADDR
DMAx_CURR_X_COUNT loaded from DMAx_X_COUNT
DMAx_CURR_Y_COUNT loaded from DMAx_Y_COUNT

The DFETCH bit in the DMAx_IRQ_STATUS register is then cleared, after which the DMA transfer begins again, as shown in Figure 5-2 on page 5-19.

Work Unit Transitions

Transitions from one work unit to the next are controlled by the SYNC bit in the DMAx_CONFIG register of the work units. In general, continuous transitions have lower latency at the cost of restrictions on changes of data format or addressed memory space in the two work units. These latency gains and data restrictions arise from the way the DMA FIFO pipeline is handled while the next descriptor is fetched. In continuous transitions (SYNC = 0), the DMA FIFO pipeline continues to transfer data to and from the peripheral or destination memory during the descriptor fetch and/or when the DMA channel is paused between descriptor chains.

Synchronized transitions (SYNC = 1), on the other hand, provide better real-time synchronization of interrupts with peripheral state and greater flexibility in the data formats and memory spaces of the two work units, at the cost of higher latency in the transition. In synchronized transitions, the DMA FIFO pipeline is drained to the destination or flushed (RX data discarded) between work units.

Work unit transitions for MDMA streams are controlled by the SYNC bit of the MDMA source channel's DMAx_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be 0. In transmit (memory read) channels, the SYNC bit of the last descriptor prior to the transition controls the transition behavior. In contrast, in receive channels, the SYNC bit of the first descriptor of the next descriptor chain controls the transition.

DMA Transmit and MDMA Source

In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work units.

If SYNC = 0, a continuous transition is selected. In a continuous transition, just after the last data item is read from memory, the following operations start in parallel:

- The interrupt (if any) is signalled.
- The DMA_DONE bit in the DMAx_IRQ_STATUS register is set.
- The next descriptor begins to be fetched.
- The final data items are delivered from the DMA FIFO to the destination memory or peripheral.

This allows the DMA to provide data from the FIFO to the peripheral "continuously" during the descriptor fetch latency period.

When SYNC = 0, the final interrupt (if enabled) occurs when the last data is read from memory. This interrupt is at the earliest time that the output memory buffer may safely be modified without affecting the previous data transmission. Up to four data items may still be in the DMA FIFO, however, and not yet at the peripheral, so the DMA interrupt should not be used as the sole means of synchronizing the shutdown or reconfiguration of the peripheral following a transmission.

If SYNC = 0 (continuous transition) on a transmit (memory read) descriptor, the next descriptor must have the same data word size, read/write direction, and source memory (internal vs. external) as the current descriptor.

SYNC = 0 selects continuous transition on a work unit in FLOW = 0 mode with interrupt enabled. The interrupt service routine may begin execution while the final data is still draining from the FIFO to the peripheral. This

is indicated by the DMA_RUN bit in the DMAx_IRQ_STATUS register; if it is 1, the FIFO is not empty yet. Do not start a new work unit with different word size or direction while DMA_RUN = 1. Further, if the channel is disabled (by writing DMAEN = 0), the data in the FIFO is lost.

SYNC = 1 selects a synchronized transition in which the DMA FIFO is first drained to the destination memory or peripheral before any interrupt is signalled and before any subsequent descriptor or data is fetched. This incurs greater latency, but provides direct synchronization between the DMA interrupt and the state of the data at the peripheral.

For example, if SYNC = 1 and DI_EN = 1 on the last descriptor in a work unit, the interrupt occurs when the final data has been transferred to the peripheral, allowing the service routine to properly switch to non-DMA transmit operation. When the interrupt service routine is invoked, the DMA_DONE bit is set and the DMA_RUN bit is cleared.

A synchronized transition also allows greater flexibility in the format of the DMA descriptor chain. If SYNC = 1, the next descriptor may have any word size or read/write direction supported by the peripheral and may come from either memory space (internal or external). This can be useful in managing MDMA work unit queues, since it is no longer necessary to interrupt the queue between dissimilar work units.

DMA Receive

In DMA receive (memory write) channels, the SYNC bit controls the handling of the DMA FIFO between descriptor chains (not individual descriptors), when the DMA channel is paused. The DMA channel pauses after descriptors with FLOW = 0 mode, and may be restarted (for example, after an interrupt) by writing the channel's DMAx_CONFIG register with DMAEN = 1.

If the SYNC bit is 0 in the new work unit's DMAx_CONFIG value, a continuous transition is selected. In this mode, any data items received into the DMA FIFO while the channel was paused are retained, and they are the first

items written to memory in the new work unit. This mode of operation provides lower latency at work unit transitions and ensures that no data items are dropped during a DMA pause, at the cost of certain restrictions on the DMA descriptors.

If the SYNC bit is 0 on the first descriptor of a descriptor chain after a DMA pause, the DMA word size of the new chain must not change from the word size of the previous descriptor chain active before the pause, unless the DMA channel is reset between chains by writing the DMAEN bit to 0 and then to 1 again.

If the SYNC bit is 1 in the new work unit's DMAx_CONFIG value, a synchronized transition is selected. In this mode, only the data received from the peripheral by the DMA channel after the write to the DMAx_CONFIG register are delivered to memory. Any prior data items transferred from the peripheral to the DMA FIFO before this register write are discarded. This provides direct synchronization between the data stream received from the peripheral and the timing of the channel restart (when the DMAx_CONFIG register is written).

For receive DMAs, the SYNC bit has no effect in transitions between work units in the same descriptor chain (that is, when the previous descriptor's FLOW mode was not 0, so that DMA channel did not pause.)

If a descriptor chain begins with a SYNC bit of 1, there is no restriction on DMA word size of the new chain in comparison to the previous chain.

The DMA word size must not change between one descriptor and the next in any DMA receive (memory write) channel within a single descriptor chain, regardless of the SYNC bit setting. In other words, if a descriptor has WNR = 1 and FLOW = 4, 6, or 7, then the next descriptor must have the same word size. For any DMA receive (memory write) channel, there is no restriction on changes of memory space (internal vs. external) between descriptors or

descriptor chains. DMA transmit (memory read) channels may have such restrictions (see "DMA Transmit and MDMA Source" on page 5-25).

Stopping DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete.

If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.

In autobuffer (FLOW = 1) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMAEN bit is 0. A write of 0 to the entire register will always terminate DMA gracefully (without DMA abort).

If a channel has been stopped abruptly by writing DMAx_CONFIG to 0 (or any value with DMAEN = 0), the user must ensure that any memory read or write accesses in the pipelines have completed before enabling the channel again. If the channel is enabled again before an "orphan" access from a previous work unit completes, the state of the DMA interrupt and FIFO is unspecified. This can generally be handled by ensuring that the core allocates several consecutive idle cycles in its usage of the relevant memory space to allow up to three pending DMA accesses to issue, plus allowing enough memory access time for the accesses themselves to complete.

DMA Errors (Aborts)

The DMA controller flags conditions that cause the DMA process to end abnormally (abort). This functionality is provided as a tool for system development and debug to detect DMA-related programming errors. DMA errors (aborts) are detected by the DMA channel module in the

cases listed below. When a DMA error occurs, the channel is immediately stopped (DMA_RUN goes to 0) and any prefetched data is discarded. In addition, a DMA_ERROR interrupt is asserted.

There is only one DMA_ERROR interrupt for the whole DMA controller, which is asserted whenever any of the channels has detected an error condition.

The DMA_ERROR interrupt handler must:

- Read each channel's DMAx_IRQ_STATUS register to look for a channel with the DMA_ERR bit set (bit 1).
- Clear the problem with that channel (for example, fix register values).
- Clear the DMA_ERR bit (write DMAx_IRQ_STATUS with bit 1 set).

The following error conditions are detected by the DMA hardware and result in a DMA abort interrupt.

- The configuration register contains invalid values:
  - Incorrect WDSIZE value (WDSIZE = b#11)
  - Bit 15 not set to 0
  - Incorrect FLOW value (FLOW = 2, 3, or 5)
  - NDSIZE value does not agree with FLOW. See Table 5-2 on page 5-31.
- A disallowed register write occurred while the channel was running. Only the DMAx_CONFIG and DMAx_IRQ_STATUS registers can be written when DMA_RUN = 1.

- An address alignment error occurred during any memory access. For example, when DMAx_CONFIG register WDSIZE = 1 (16-bit) but the least significant bit (LSB) of the address is not equal to b#0, or when WDSIZE = 2 (32-bit) but the two LSBs of the address are not equal to b#00.
- A memory space transition was attempted (internal-to-external or vice versa). For example, the value in the DMAx_CURR_ADDR register or DMAx_CURR_DESC_PTR register crossed a memory boundary.
- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (signaled by the external memory interface).

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signaled for these situations:

- DMAx_CONFIG direction bit (WNR) does not agree with the direction of the mapped peripheral.
- DMAx_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMAx_CONFIG word size (WDSIZE) is not supported by the mapped peripheral. See Table 5-2 on page 5-31.
- DMAx_CONFIG word size in source and destination of the MDMA stream are not equal.
- Descriptor chain indicates data buffers that are not in the same internal/external memory space.
- In 2-D DMA, X_COUNT = 1

Table 5-2. Legal NDSIZE Values

FLOW	NDSIZE	Note
0	0	
1	0	
4	0 < NDSIZE ≤ 7	Descriptor array, no descriptor pointer fetched
6	0 < NDSIZE ≤ 8	Descriptor list, small descriptor pointer fetched
7	0 < NDSIZE ≤ 9	Descriptor list, large descriptor pointer fetched

DMA Control Commands

Advanced peripherals, such as an Ethernet MAC module, are capable of managing some of their own DMA operations, thus dramatically improving real-time performance and relieving control and interrupt demands on the Blackfin processor core. These peripherals may communicate to the DMA controller using DMA control commands, which are transmitted from the peripheral to the associated DMA channel over internal DMA request buses. Refer to "Unique Information for the ADSP-BF59x Processor" on page 5-101 to determine if DMA control commands are applicable to a particular product.

The request buses consist of three wires per DMA-management-capable peripheral. The DMA control commands extend the set of operations available to the peripheral beyond the simple "request data" command used by peripherals in general.

While these DMA control commands are not visible to or controllable by the user, their use by a peripheral has implications for the structure of the DMA transfers which that peripheral can support. It is important that application software be written to comply with certain restrictions regarding work units and descriptor chains (described later in this section) so that the peripheral operates properly whenever it issues DMA control commands.

MDMA channels do not service peripherals and therefore do not support DMA control commands. The DMA control commands are shown in Table 5-3.

Table 5-3. DMA Control Commands

Code	Name	Description
000	NOP	No operation
001	Restart	Restarts the current work unit from the beginning
010	Finish	Finishes the current work unit and starts the next
011	-	Reserved
100	Req Data	Typical DMA data request
101	Req Data Urgent	Urgent DMA data request
110	-	Reserved
111	-	Reserved

Additional information for the control commands includes:

- **Restart**

  The `Restart` command causes the current work unit to interrupt processing and start over, using the addresses and counts from `DMAx_START_ADDR`, `DMAx_X_COUNT`, and `DMAx_Y_COUNT`. No interrupt is signalled.

  If a channel programmed for transmit (memory read) receives a `Restart` command, the channel momentarily pauses while any pending memory reads initiated prior to the `Restart` command are completed.

  During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel resets its counters and FIFO and starts prefetch reads from memory. DMA data requests from the

peripheral are granted as soon as new prefetched data is available in the DMA FIFO. The peripheral can thus use the `Restart` command to re-attempt a failed transmission of a work unit.

If a channel programmed for receive (memory write) receives a `Restart` command, the channel stops writing to memory, discards any data held in its DMA FIFO, and resets its counters and FIFO. As soon as this initialization is complete, the channel again grants DMA write requests from the peripheral. The peripheral can thus use the `Restart` command to abort transfer of received data into a work unit and re-use the memory buffer for a later data transfer.

- **Finish**

  The `Finish` command causes the current work unit to terminate and move on to the next work unit. An interrupt is signalled as usual, if selected by the `DI_EN` bit. The peripheral can thus use the `Finish` command to partition the DMA stream into work units on its own, perhaps as a result of parsing the data currently passing though its supported communication channel, without direct real-time control by the processor.

  If a channel programmed for transmit (memory read) receives a `Finish` command, the channel momentarily pauses while any pending memory reads initiated prior to the `Finish` command are completed. During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel signals an interrupt (if enabled), and begins fetching the next descriptor (if any). DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO.

  If a channel programmed for receive (memory write) receives a `Finish` command, the channel stops granting new DMA requests while it drains its FIFO. Any DMA data received by the DMA controller prior to the `Finish` command is written to memory. When

the FIFO reaches an empty state, the channel signals an interrupt (if enabled) and begins fetching the next descriptor (if any). Once the next descriptor has been fetched, the channel initializes its FIFO and then resumes granting DMA requests from the peripheral.

- **Request Data**

  The `Request Data` command is identical to the DMA request operation of peripherals that are not DMA-management-capable.

- **Request Data Urgent**

  The `Request Data Urgent` command behaves identically to the DMA `Request` command, except that the DMA channel performs its memory accesses with urgent priority while it is asserted. This includes both data and descriptor-fetch memory accesses. A DMA-management-capable peripheral might use this command if an internal FIFO is approaching a critical condition.

Restrictions

The proper operation of the 4-location DMA channel FIFO leads to certain restrictions in the sequence of DMA control commands.

Transmit Restart or Finish

No `Restart` or `Finish` command may be issued by a peripheral to a channel configured for memory read unless the peripheral has already performed at least one DMA transfer in the current work unit and the current work unit has more than four items remaining in `DMAx_CURR_X_COUNT`/ `DMAx_CURR_Y_COUNT` (thus not yet read from memory). Otherwise, the current work unit may already have completed memory operations and can no longer be restarted or finished properly.

If the `DMAx_CURR_X_COUNT`/ `DMAx_CURR_Y_COUNT` value of the current work unit is sufficiently large that it is always at least five more than the

maximum data count prior to any Restart or Finish command, the above restriction is satisfied. This implies that any work unit which might be managed by Restart or Finish commands must have DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing at least five data items.

Particularly if the DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT registers are programmed to 0 (representing 65,536 transfers, the maximum value) the channel will operate properly for 1-D work units up to 65,531 data items or 2-D work units up to 4,294,967,291 data items.

Receive Restart or Finish

No Restart or Finish command may be issued by a peripheral to a channel configured for memory write unless either the peripheral has already performed at least five DMA transfers in the current work unit or the previous work unit was terminated by a Finish command and the peripheral has performed at least one DMA transfer in the current work unit. If five data transfers have been performed, then at least one data item has been written to memory in the current work unit, which implies that the current work unit's descriptor fetch completed before the data grant of the fifth item. Otherwise, if less than five data items have been transferred, it is possible that all of them are still in the DMA FIFO and the previous work unit is still in the process of completion and transition between work units.

Similarly, if a Finish command ended the previous work unit and at least one subsequent DMA data transfer has occurred, then the fact that the DMA channel issued the grant guarantees that the previous work unit has already completed the process of draining its data to memory and transitioning to the new work unit.

If a peripheral terminates all work units with the Finish opcode (effectively assuming responsibility for all work unit boundaries for the DMA channel), then the peripheral need only ensure that it performs a single transfer in each work unit before any restart or finish. This requires, however, that the user programs the descriptors for all work units managed by

the channel with DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing more data items than the maximum work unit size that the peripheral will encounter. For example, DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values of 0 allow the channel to operate properly on 1-D work units up to 65,535 data items and 2-D work units up to 4,294,967,295 data items.

Handshaked Memory DMA Operation

Handshaked memory DMA operation is not available for all products. Refer to the "Unique Information for the ADSP-BF59x Processor" on page 5-101 to determine whether this feature applies to this product.

Each DMARx input has its own set of control and status registers. Handshake operation for MDMA0 is enabled by the HMDMAEN bit in the HMDMA0_CONTROL register. Similarly, the HMDMAEN bit in the HMDMA1_CONTROL register enables handshake mode for MDMA1.

It is important to understand that the handshake hardware works completely independently from the descriptor and autobuffer capabilities of the MDMA, allowing most flexible combinations of logical data organization vs. data portioning as required by FIFO depths, for example. If, however, the connected device requires certain behavior of the address lines, these must be controlled by traditional DMA setup.

The HMDMA unit controls only the destination (memory write) channel of the memory DMA. The source channel (memory-read side) fills the 8-deep DMA buffers immediately after the receive side is enabled and issues eight read commands.

The HMDMAx_BCINIT registers control how many data transfers are performed upon every DMA request. If set to one, the peripheral can time every individual data transfer. If greater than one, the peripheral must have sufficient buffer size to provide or consume the number of words programmed. Once the transfer has been requested, no further handshake can hold off the DMA from transferring the entire block, except by

stalling the EBIU accesses by the ARDY signal. Nevertheless, the peripheral may request a block transfer before the entire buffer is available by simply taking the minimum transfer time based on wait-state settings into consideration.

The block count defines how many data transfers are performed by the MDMA engine. A single DMA transfer can cause two read or write operations on the EBIU port if the transfer word size is set to 32-bit in the MDMA_yy_CONFIG register (WDSIZE = b#10).

Since the block count registers are 16 bits wide, blocks can group up to 65,535 transfers.

Once a block transfer has been started, the HMDMAx_BCOUNT registers return the remaining number of transfers to complete the current block. When the complete block has been processed, the HMDMAx_BCOUNT register returns zero. Software can force a reload of the HMDMAx_BCOUNT from the HMDMAx_BCINIT register even during normal operation by setting the RBC bit in the HMDMAx_CONTROL register. Set RBC when the HMDMA module is already active, but only when the MDMA is not enabled.

Pipelining DMA Requests

The device mastering the DMA request lines is allowed to request additional transfers even before the former transfer has completed. As long as the device can provide or consume sufficient data it is permitted to pulse the DMARx inputs multiple times.

The HMDMAx_ECOUNT registers are incremented every time a significant edge is detected on the respective DMARx input, and they are decremented when the MDMA completes the block transfer. These read-only registers use a 16-bit twos-complement data representation: if they return zero, all requested block transfers have been performed. A positive value signals up to 32767 requests that haven't been served yet and indicates that the MDMA is currently processing. Negative values indicate the number of

DMA requests that will be ignored by the engine. This feature restrains initial pulses on the DMARx inputs at startup.

The HMDMAx_ECINIT registers reload the HMDMAx_ECOUNT registers every time the handshake mode is enabled (when the HMDMAEN bit changes from 0 to 1). If the initial edge count value is 0, the handshake operation starts with a settled request budget. If positive, the engine starts immediately transferring the programmed number (up to 32767) of blocks once enabled, even without detecting any activity on the DMARx pins. If negative, the engine will disregard the programmed number (up to 32768) significant edges on the DMARx inputs before starting normal operation.

Figure 5-3 illustrates how an asynchronous FIFO could be connected. In such a scenario the REP bit should be cleared to let the DMARx request pin listen to falling edges. The Blackfin processor does not evaluate the full flag such FIFOs usually provide because asynchronous polling of that signal would reduce the system throughput drastically. Moreover, the processor first fills the FIFO by initializing the HMDMAx_ECINIT register to 1024, which equals the depth of the FIFO. Once enabled, the MDMA automatically transmits 1024 data words. Afterward it continues to transmit only if the FIFO is emptied by its read strobe again. Most likely, the HMDMAx_BCINIT register is programmed to 1 in this case.

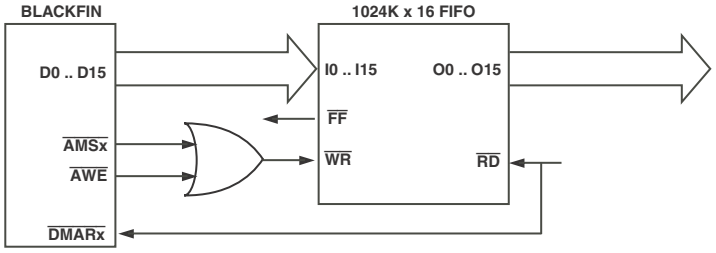


Figure 5-3. Transmit DMA Example Connection

In the receive example shown in Figure 5-4, the Blackfin processor again does not use the FIFO's internal control mechanism. Rather than testing the empty flag, the processor counts the number of data words available in

the FIFO in its own HMDMAx_ECOUNT register. Theoretically, the MDMA could immediately process data as soon as it is written into the FIFO by the write strobe, but the fast MDMA engine would read out the FIFO quickly and stall soon if the FIFO was not promptly filled with new data . Streaming applications can balance the FIFO so that the producer is never held off by a full FIFO and the consumer is never held by an empty FIFO. This is accomplished by filling the FIFO halfway and then letting both consumer and producer run at the same speed. In this case the HMDMAx_ECINIT register can be written with a negative value, which corresponds to half the FIFO depth. Then, the MDMA does not start consuming data as long as the FIFO is not half-filled.

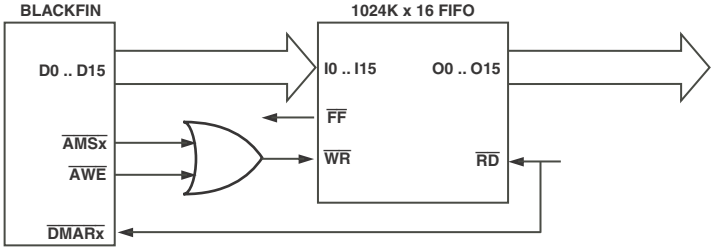


Figure 5-4. Receive DMA Example Connection

On internal system buses, memory DMA channels have lower priority than other DMAs. In busy systems, the memory DMAs may tend to starve. As this is not acceptable when transferring data through high-speed FIFOs, the handshake mode provides a high-water functionality to increase the MDMA's priority. With the UTE bit in the HMDMAx_CONTROL register set, the MDMA gets higher priority as soon as a (positive) value in the HMDMAx_ECOUNT register becomes higher than the threshold held by the HMDMAx_ECURGENT register.

HMDMA Interrupts

In addition to the normal MDMA interrupt channels, the handshake hardware provides two new interrupt sources for each DMARx input. The

HMDMAx_CONTROL registers provide interrupt enable and status bits. The interrupt status bits require a write-1-to-clear operation to cancel the interrupt request.

The block done interrupt signals that a complete MDMA block, as defined by the HMDMAx_BCINIT register, has been transferred (when the HMDMAx_BCOUNT register decrements to zero). While the BDIE bit enables this interrupt, the MBDI bit can gate it until the edge count also becomes zero, meaning that all requested MDMA transfers have been completed.

The overflow interrupt is generated when the HMDMA_ECOUNT register overflows. Since it can count up to 32767, which is much more than most peripheral devices can support, the Blackfin processor has another threshold register called HMDMA_ECOVERFLOW. It resets to 0xFFFF and should be written with any positive value by the user before enabling the function by the OIE bit. Then, the overflow interrupt is issued when the value of the HMDMA_ECOUNT register exceeds the threshold in the HMDMA_ECOVERFLOW register.

DMA Performance

The DMA system is designed to provide maximum throughput per channel and maximum utilization of the internal buses, while accommodating the inherent latencies of memory accesses.

The Blackfin architecture features several mechanisms to customize system behavior for best performance. This includes DMA channel prioritization, traffic control, and priority treatment of bursted transfers. Nevertheless, the resulting performance of a DMA transfer often depends on application-level circumstances. For best performance consider the following system software architecture questions.

- What is the required DMA bandwidth?
- Which DMA transfers have real-time requirements and which do not?

- How heavily is the DMA controller competing with the core for on-chip and off-chip resources?
- How often do competing DMA channels require the bus systems to alter direction?
- How often do competing DMA or core accesses cause the SDRAM to open different pages?
- Is there a way to distribute DMA requests nicely over time?

A key feature of the DMA architecture is the separation of the activity on the DMA access bus (DAB) used by the peripherals from the activity on the buses between the DMA and memory. For DMA to/from on-chip memory the DMA core bus (DCB) is used, and the DMA external bus (DEB) is used for DMA transfers with off-chip memory. The “Chip Bus Hierarchy” chapter explains the bus architecture.

Each peripheral DMA channel has its own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

DMA Throughput

Each peripheral DMA channel has a maximum transfer rate of one 16-bit word per two system clocks in either direction. Like the DAB and DEB buses, the DMA controller resides in the SCLK domain. The controller synchronizes accesses to and from the DCB bus, which runs at the CCLK rate.

Each memory DMA channel has a maximum transfer rate of one 16-bit word per system clock (SCLK) cycle.

When the traffic on all DMA channels is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of one 16-bit transfer per system clock.

Some considerations which limit the actual performance include:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example when both the core and the DMA access the same L1 bank, when SDRAM pages need to be opened/closed, or when cache lines are filled.
- Direction changes from `RX` to `TX` on the DAB bus impose a one `SCLK` cycle delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.
- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.
- MMR accesses to DMA registers other than `DMAx_CONFIG`, `DMAx_IRQ_STATUS`, or `DMAx_PERIPHERAL_MAP` stall all DMA activity for one cycle per 16-bit word transferred. In contrast, MMR accesses to the control/status registers do not cause stalls or wait states.
- Reads from DMA registers other than control/status registers use one PAB bus wait state, delaying the core for several core clocks.

- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.
- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMAEN changes from 0 to 1 or when the SYNC bit is set in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or autobuffering.

Efficiency loss caused by excessive direction changes (thrashing) can be minimized by the processor's traffic control features, described in the next section.

The MDMA channels are clocked by SCLK. If the source and destination are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. For example DMA typically runs at 2/3 of the system clock rate when the core-to-system clock ratio is 2:1. At higher clock ratios, full bandwidth is maintained.

If the source and destination are in the same memory space (both internal or both external), the MDMA stream typically prefetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is dependent on traffic, and is equal to three plus the memory latency at the

DMA in SCLKs (which is typically seven for internal transfers and six for external transfers).

Memory DMA Timing Details

When the destination DMAx_CONFIG register is written, MDMA operation starts after a latency of three SCLK cycles.

If either MDMA channel has been selected to use descriptors, the descriptors are fetched from memory. The destination channel descriptors are fetched first. Then the source MDMA channel begins fetching data from the source buffer, after a latency of four SCLK cycles after the last descriptor word is returned from memory. Due to memory pipelining, this is typically eight SCLK cycles after the fetch of the last descriptor word. The resulting data is deposited in the MDMA channel's 8-location FIFO. After a latency of two SCLK cycles, the destination MDMA channel begins writing data to the destination memory buffer.

Static Channel Prioritization

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the PMAP field in the DMAx_PERIPHERAL_MAP registers. The memory DMA streams are always lower static priority than the peripherals, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers.

Temporary DMA Urgency

Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals

is not too large a fraction of the total, then all peripherals' requests should be granted as required.

Occasionally, instantaneous DMA traffic might exceed the available bandwidth, causing congestion. This may occur if L1 or external memory is temporarily stalled, perhaps for an SDRAM page swap or a cache line fill. Congestion might also occur if one or more DMA channels initiates a flurry of requests, perhaps for descriptor fetches or to fill a FIFO in the DMA or in the peripheral.

If congestion persists, lower priority DMA peripherals may become starved for data. Even though the peripheral's priority is low, if the necessary data transfer does not take place before the end of the peripheral's regular interval, system failure may result. To minimize this possibility, the DMA unit detects peripherals whose need for data has become urgent, and preferentially grants them service at the highest priority.

A DMA channel's request for memory service is defined as urgent if both:

- The channel's FIFO is not ready for a DAB bus transfer (that is, a transmit FIFO is empty or a receive FIFO is full), and
- The peripheral is asserting its DMA request line.

Descriptor fetches may be urgent if they are necessary to initiate or continue a DMA work unit chain for a starving peripheral.

DMA requests from an MDMA channel become urgent when handshaked operation is enabled and the DMARx edge count exceeds the value stored in the HMDMAx_ECURGENT register. If handshaked operation is disabled, software can control urgency of requests directly by altering the DRQ bit field in the HMDMAx_CONTROL register.

When one or more DMA channels express an urgent memory request, two events occur:

- All non-urgent memory requests are decreased in priority by 32, guaranteeing that only an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external). All prior incomplete memory transfers ahead of it in that memory system are also marked for expedited processing. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

The preferential handling of urgent DMA transfers is completely automatic. No user controls are required for this function to operate.

Memory DMA Priority and Scheduling

All MDMA operations have lower precedence than any peripheral DMA operations. MDMA thus makes effective use of any memory bandwidth unused by peripheral DMA traffic.

By default, when more than one MDMA stream is enabled and ready, only the highest priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, the `MDMA_ROUND_ROBIN_PERIOD` may be programmed to select each stream in turn for a fixed number of transfers.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round-robin scheme. This is selected by programming the `MDMA_ROUND_ROBIN_PERIOD` field in the `DMA_TC_PER` register (see "Static Channel Prioritization" on page 5-44).

If this field is set to 0, then MDMA is scheduled by fixed priority. MDMA stream 0 takes precedence over MDMA stream 1 whenever stream 0 is ready to perform transfers. Since an MDMA stream is typically capable of transferring data on every available cycle, this could cause MDMA stream 1 traffic to be delayed for an indefinite time until any and all MDMA stream 0 operations are completed. This scheme could be appropriate in systems where low duration but latency-sensitive data buffers need to be moved immediately, interrupting long duration, low priority background transfers.

If the MDMA_ROUND_ROBIN_PERIOD field is set to some nonzero value in the range $1 \leq P \leq 31$, then a round-robin scheduling method is used. The two MDMA streams are granted bus access in alternation in bursts of up to P data transfers. This could be used in systems where two transfer processes need to coexist, each with a guaranteed fraction of the available bandwidth. For example, one stream might be programmed for internal-to-external moves while the other is programmed for external-to-internal moves, and each would be allocated approximately equal data bandwidth.

In round-robin operation, the MDMA stream selection at any time is either “free” or “locked.” Initially, the selection is free. On any free cycle available to MDMA (when no peripheral DMA accesses take precedence), if either or both MDMA streams request access, the higher precedence stream will be granted (stream 0 in case of conflict), and that stream’s selection is then “locked.” The MDMA_ROUND_ROBIN_COUNT counter field in the DMA_TC_CNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of one, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not

ready to perform a transfer, then no transfer is performed, and the stream selection unlocks and becomes free again on the next cycle.

If round-robin operation is used when only one MDMA stream is active, one idle cycle will occur for each P MDMA data cycles, slightly lowering the bandwidth by a factor of 1/(P+1). However if both MDMA streams are used, memory DMA can operate continuously with zero additional overhead for alternation of streams . (Other than overhead cycles normally associated with reversal of read/write direction to memory). By selection of various round-robin period values P, which limit how often the MDMA streams alternate, maximal transfer efficiency can be maintained.

Traffic Control

In the Blackfin DMA architecture, there are two completely separate but simultaneous prioritization processes—the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are requesting DMA via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (SCLK). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see “Temporary DMA Urgency” on page 5-44.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same direction transfers together. The DMA block provides a traffic control

mechanism controlled by the DMA_TC_PER and DMA_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMA_TC_CNT register. See "Memory DMA Priority and Scheduling" on page 5-46.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out or traffic stops or changes direction on its own. When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower priority channel 5 is going with traffic and higher priority channel 3 is going against traffic, then channel 3's effective priority becomes 19, and channel 5 would be granted instead. If, on the next cycle, only channels 3 and 6 were requesting DAB transfers, and these transfer requests were both against traffic, then their effective priorities would become 19 and 22, respectively. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required for the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the DMA_TC_PER register to 0x0000.

Programming Model

Several synchronization and control methods are available for use in development of software tasks which manage peripheral DMA and memory DMA (see also "Memory DMA" on page 5-6). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each peripheral DMA and memory DMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the `DMAx_IRQ_STATUS` register.

Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of `DMAx_IRQ_STATUS`, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel's interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note, since every interrupt channel

has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Due to DMA FIFOs and DMA/memory pipelining, polling of the DMAx_CURR_ADDR, DMAx_CURR_DESC_PTR, or DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT registers is not recommended for precisely synchronizing DMA with data processing. The current address, pointer, and count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation would first be visible to the core by memory read or write instructions. For example, in a DMA memory write operation to external memory, assume a DMA write by channel A is initiated that causes the SDRAM to perform a page open operation which takes many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but will be stalled behind the slow operation of channel A. Software monitoring of channel B, based on examination of the DMAx_CURR_ADDR register contents, would not safely conclude whether the memory location pointed to by channel B's DMAx_CURR_ADDR register has or has not been written.

If allowances are made for the lengths of the DMA/memory pipeline, polling of the current address, pointer, and count registers can permit loose synchronization of DMA with software. The depth of the DMA FIFO is four locations (either four 8- or 16-bit data elements, or two 32-bit data elements) for a peripheral DMA channel, and eight locations (four 32-bit data elements) for an MDMA FIFO. The DMA will not advance current address/pointer/count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished).

Additionally, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements. The length of the DMA and external bus interface unit (EBIU) pipelines is approximately three data elements, when measured from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO

length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. This value is a maximum because the DMA/memory pipeline may include traffic from other DMA channels.

For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its DMAx_CURR_X_COUNT register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. Since the total pipeline length is no greater than the sum of four (for the peripheral DMA FIFO) plus six (for the DMA/memory pipeline) or ten data elements, it is safe to conclude that the DMA transfer of the first 30 (40-10) data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel's DMAx_IRQ_STATUS register to confirm completion of DMA, rather than polling current address/pointer/count registers. When the DMA system issues an interrupt or changes a DMAx_IRQ_STATUS bit, it guarantees that the last memory operation of the work unit has been completed and will definitely be visible to processor code. For memory read DMA, the final memory read data will have been safely received in the DMA's FIFO. For memory write DMA, the DMA unit will have received an acknowledgement from L1 memory, or the EBIU, that the data has been written.

The following examples show methods of synchronizing software with several different styles of DMA.

Single-Buffer DMA Transfers

Synchronization is simple if a peripheral's DMA activity consists of isolated transfers of single buffers. DMA activity is initiated by software writes to the channel's control registers. The user may choose to use a single descriptor in memory, in which case the software only needs to write the DMAx_CONFIG and the DMAx_NEXT_DESC_PTR registers. Alternatively, the

user may choose to write all the MMR registers directly from software, ending with the write to the DMAx_CONFIG register.

The simplest way to signal completion of DMA is by an interrupt. This is selected by the DI_EN bit in the DMAx_CONFIG register, and by the necessary setup of the system interrupt controller. If no interrupt is desired, the software can poll for completion by reading the DMAx_IRQ_STATUS register and testing the DMA_RUN bit. If this bit is zero, the buffer transfer has completed.

Continuous Transfers Using Autobuffering

If a peripheral's DMA data consists of a steady, periodic stream of signal data, DMA autobuffering (FLOW = 1) may be an effective option. Here, DMA is transferred from or to a memory buffer with a circular addressing scheme, using either one- or two-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

- 1-D interrupt-driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.
- 2-D interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set DI_SEL = 1 in DMAx_CONFIG) to be signaled at the completion of each DMA inner loop. In this way, a traditional double buffer or "ping-pong" scheme can be implemented. For example, two 512-word sub-buffers inside a 1K-word buffer could be used to receive 16-bit peripheral data with the settings in Table 5-4.

Table 5-4. 2-D Interrupt-Driven Double Buffered DMA

Register	Setting
DMAx_START_ADDR	buffer base address
DMAx_CONFIG	0x10D7 (FLOW = 1, DI_EN = 1, DI_SEL = 1, DMA2D = 1, WDSIZE = b#01, WNR = 1, DMAEN = 1)
DMAx_X_COUNT	512
DMAx_X_MODIFY	2 for 16-bit data
DMAx_Y_COUNT	2 for two sub-buffers
DMAx_Y_MODIFY	2 same as DMAx_X_MODIFY for contiguous sub-buffers

- 2-D polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2-D multibuffer synchronization scheme may be used. For example, assume receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2-D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with the settings in Table 5-5.

Table 5-5. 2-D Polled DMA

Register	Setting
DMAx_START_ADDR	buffer base address
DMAx_CONFIG	0x101B (FLOW = 1, DI_EN = 0, DMA2D = 1, WDSIZE = b#10, WNR = 1, DMAEN = 1)
DMAx_X_COUNT	16
DMAx_X_MODIFY	4 for 32-bit data
DMAx_Y_COUNT	4 for four sub-buffers
DMAx_Y_MODIFY	4 same as DMAx_X_MODIFY for contiguous sub-buffers

- The synchronization core might read `DMAx_Y_COUNT` to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of

DMAx_Y_COUNT shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.

- 1-D unsynchronized FIFO—if a system's design guarantees that the processing of a peripheral's data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1-D autobuffer mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1-D or 2-D arrays. For example, if a packet of data is to be transmitted from several different locations in memory (a header from one location, a payload from a list of several blocks of memory managed by a memory pool allocator, and a small trailer containing a checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list by selecting the appropriate FLOW setting in DMAx_CONFIG.

The software can synchronize with the progress of the structure's transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header's descriptor and for the trailer's descriptor, but not for the payload blocks' descriptors.

It is important to remember the meaning of the various fields in the DMAx_CONFIG descriptor elements when building a list or array of DMA descriptors. In particular:

- The lower byte of DMAx_CONFIG specifies the DMA transfer to be performed by the *current* descriptor (for example 2-D interrupt-enable mode)

- The upper byte of DMAx_CONFIG specifies the format of the *next* descriptor in the chain. The NDSIZE and FLOW fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is being restarted, both bytes of the DMAx_CONFIG value written to the DMA channel's DMAx_CONFIG register should correspond to the current descriptor. At a minimum, the FLOW, NDSIZE, WNR, and DMAEN fields must all agree with the current descriptor. The WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be taken from the DMAx_CONFIG value in the descriptor read from memory. The field values initially written to the register are ignored. See "Initializing Descriptors in Memory" on page 5-93 in the "Programming Examples" section for information on how descriptors can be set up.

Descriptor Queue Management

A system designer might want to write a DMA manager facility which accepts DMA requests from other software. The DMA manager software does not know in advance when new work requests will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's NDPH and NDPL members point to the next descriptor, and the last descriptor points back to the first.

The code that writes into this descriptor list could use the processor's circular addressing modes (Ix, Lx, Mx, and Bx registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the NDPH and NDPL members of each descriptor could even be written once at startup and skipped over as each descriptor's new contents are written.

The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

- Interrupt on every descriptor
- Interrupt minimally - only on the last descriptor

Descriptor Queue Using Interrupts on Every Descriptor

In this system, the DMA manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should only be used if system design can guarantee that each interrupt event will be serviced separately (no interrupt overrun).

To maintain synchronization of the descriptor queue, the non-interrupt software maintains a count of descriptors added to the queue, while the interrupt handler maintains a count of completed descriptors removed from the queue. The counts are equal only when the DMA channel is paused after having processed all the descriptors.

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a `DMAx_CONFIG` value with a `FLOW` value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts are equal), the software increments its count and then starts the DMA unit by writing the new descriptor's `DMAx_CONFIG` value to the DMA channel's `DMAx_CONFIG` register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's `DMAx_CONFIG` value so that its upper half (`FLOW` and `NDSIZE`) now describes the newly queued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old `DMAx_CONFIG` value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel's DMAx_IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was queued too late (the modification of the next-to-last descriptor's DMAx_CONFIG element occurred after that element was read into the DMA unit). In this case, the interrupt handler should write the DMAx_CONFIG value appropriate for the last descriptor to the DMA channel's DMAx_CONFIG register, increment the completed descriptor count, and exit.

Again, this system can fail if the system's interrupt latencies are large enough to cause any of the channel's DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptors' interrupts would need to be complex, performing several MMR accesses to ensure robust operation. In such a system environment, a minimal interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an "active" and a "waiting" portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software's non-interrupt code fills in a new descriptor's contents and adds it to the waiting portion of the queue. The descriptor's DMAx_CONFIG word should have a FLOW value of zero. If more than one request is received before the DMA queue completion interrupt occurs, the non-interrupt code should queue later descriptors, forming a waiting portion of the queue that is disconnected from the active portion of the queue being processed by the DMA unit. In other words, all but the last active descriptors contain FLOW values ≥ 4 and

have no interrupt enable set, while the last active descriptor contains a FLOW of 0 and an interrupt enable bit DI_EN set to 1. Also, all but the last waiting descriptors contain FLOW values ≥ 4 and no interrupt enables set, while the last waiting descriptor contains a FLOW of 0 and an interrupt enable bit set. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler with a single DMAx_CONFIG register write.

After queuing a new waiting descriptor, the non-interrupt software should leave a message for its interrupt handler in a memory mailbox location containing the desired DMAx_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting).

Once processing by the DMA unit has started, it is critical that the software not directly modify the contents of the active descriptor queue unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software would never modify descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

When a DMA queue completion interrupt is received, the interrupt handler reads the mailbox from the non-interrupt software and writes the value in it to the DMA channel's DMAx_CONFIG register. This single register write restarts the queue, effectively transforming the waiting queue to an active queue. The interrupt handler should then pass a message back to the non-interrupt software indicating the location of the last descriptor accepted into the active queue. If, on the other hand, the interrupt handler reads its mailbox and finds a DMAx_CONFIG value of zero, indicating there is no more work to perform, then it should pass an appropriate message (for example zero) back to the non-interrupt software indicating that the queue has stopped. This simple handler should be able to be coded in a very small number of instructions.

The non-interrupt software which accepts new DMA work requests needs to synchronize the activation of new work with the interrupt handler. If the queue has stopped (the mailbox from the interrupt software is zero), the non-interrupt software is responsible for starting the queue (writing the first descriptor's DMAx_CONFIG value to the channel's DMAx_CONFIG register). If the queue is not stopped, the non-interrupt software must not write to the DMAx_CONFIG register (which would cause a DMA error). Instead the descriptor should queue to the waiting queue, and update its mailbox directed to the interrupt handler.

Software Triggered Descriptor Fetches

If a DMA has been stopped in FLOW = 0 mode, the DMA_RUN bit in the DMAx_IRQ_STATUS register remains set until the content of the internal DMA FIFOs has been completely processed. Once the DMA_RUN bit clears, it is safe to restart the DMA by simply writing again to the DMAx_CONFIG register. The DMA sequence is repeated with the previous settings.

Similarly, a descriptor-based DMA sequence that has been stopped temporarily with a FLOW = 0 descriptor can be continued with a new write to the configuration register. When the DMA controller detects the FLOW = 0 condition by loading the DMACFG field from memory, it has already updated the next descriptor pointer, regardless of whether operating in descriptor array mode or descriptor list mode.

The next descriptor pointer remains valid if the DMA halts and is restarted. As soon as the DMA_RUN bit clears, software can restart the DMA and force the DMA controller to fetch the next descriptor. To accomplish this, the software writes a value with the DMAEN bit set and with proper values in the FLOW and NDSIZE fields into the configuration register. The next descriptor is fetched if FLOW equals 0x4, 0x6, or 0x7. In this mode of operation, the NDSIZE field should at least span up to the DMACFG field to overwrite the configuration register immediately.

One possible procedure is:

1. Write to DMAx_NEXT_DESC_PTR
2. Write to DMAx_CONFIG with
    - FLOW = 0x8
    - NDSIZE ≥ 0xA
    - DI_EN = 0
    - DMAEN = 1
3. Automatically fetched DMACFG has
    - FLOW = 0x0
    - NDSIZE = 0x0
    - SYNC = 1 (for transmitting DMAs only)
    - DI_EN = 1
    - DMAEN = 1
4. In the interrupt routine, repeat step 2. The DMAx_NEXT_DESC_PTR is updated by the descriptor fetch.

To avoid polling of the DMA_RUN bit, set the SYNC bit in case of memory read DMAs (DMA transmit or MDMA source).

If all DMACFG fields in a descriptor chain have the FLOW and NDSIZE fields set to zero, the individual DMA sequences do not start until triggered by software. This is useful when the DMAs need to be synchronized with other events in the system, and it is typically performed by interrupt service routines. A single MMR write is required to trigger the next DMA sequence.

Especially when applied to MDMA channels, such scenarios play an important role. Usually, the timing of MDMAs cannot be controlled (see

"Handshaked Memory DMA Operation" on page 5-36). By halting descriptor chains or rings this way, the whole DMA transaction can be broken into pieces that are individually triggered by software.

Source and destination channels of a MDMA may differ in descriptor structure. However, the total work count must match when the DMA stops. Whenever a MDMA is stopped, destination and source channels should both provide the same FLOW = 0 mode after exactly the same number of words. Accordingly, both channels need to be started afterward.

Software-triggered descriptor fetches are illustrated in Listing 5-7 on page 5-96. MDMA channels can be paused by software at any time by writing a 0 to the DRQ bit field in the HMDMAx_CONTROL register. This simply disables the self-generated DMA requests, whether or not the HMDMA is enabled.

DMA Registers

DMA registers fall into three categories:

- DMA channel registers
- Handshaked MDMA registers
- Global DMA traffic control registers

DMA Channel Registers

A processor features up to twelve peripheral DMA channels and two channel pairs for memory DMA. All channels have an identical set of registers as summarized in Table 5-6.

Table 5-6 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register,

the register category, and where applicable, the corresponding name for the data element in a DMA descriptor.

Table 5-6. Generic Names of the DMA Memory-mapped Registers

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x00	NEXT_DESC_PTR	Link pointer to next descriptor	Parameter	NDPH (upper 16 bits), NDPL (lower 16 bits)
0x04	START_ADDR	Start address of current buffer	Parameter	SAH (upper 16 bits), SAL (lower 16 bits)
0x08	CONFIG	DMA Configuration register, including enable bit	Parameter	DMACFG
0x0C	Reserved	Reserved		
0x10	X_COUNT	Inner loop count	Parameter	XCNT
0x14	X_MODIFY	Inner loop address increment, in bytes	Parameter	XMOD
0x18	Y_COUNT	Outer loop count (2-D only)	Parameter	YCNT
0x1C	Y_MODIFY	Outer loop address increment, in bytes	Parameter	YMOD
0x20	CURR_DESC_PTR	Current descriptor pointer	Current	N/A
0x24	CURR_ADDR	Current DMA address	Current	N/A
0x28	IRQ_STATUS	Interrupt status register contains completion and DMA error interrupt status and channel state (run/fetch/paused)	Control/ Status	N/A

Table 5-6. Generic Names of the DMA Memory-mapped Registers (Continued)

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x2C	PERIPHERAL_MAP	Peripheral to DMA channel mapping contains a 4-bit value specifying the peripheral associated with this DMA channel (read-only for MDMA channels)	Control/ Status	N/A
0x30	CURR_X_COUNT	Current count (1-D) or intra-row X count (2-D); counts down from X_COUNT	Current	N/A
0x34	Reserved	Reserved		
0x38	CURR_Y_COUNT	Current row count (2-D only); counts down from Y_COUNT	Current	N/A
0x3C	Reserved	Reserved		

Channel-specific register names are composed of a prefix and the generic MMR name shown in Table 5-6. For peripheral DMA channels the prefix "DMAx_" is used, where "x" stands for a channel number between 0 and 11. For memory DMA channels, the prefix is "MDMA_yy_", where "yy" stands for either "D0", "S0", "D1", or "S1" to indicate destination and source channel registers of MDMA0 and MDMA1. For example the peripheral DMA channel 6 configuration register is called DMA6_CONFIG. The register for the MDMA1 source channel is called MDMA_S1_CONFIG.

The generic MMR names shown in Table 5-6 are not actually mapped to resources in the processor.

For convenience, discussions in this chapter use generic (non-peripheral specific) DMA and memory DMA register names.

DMA channel registers fall into three categories.

- Parameter registers such as DMAx_CONFIG and DMAx_X_COUNT that can be loaded directly from descriptor elements as shown in Table 5-6
- Current registers such as DMAx_CURR_ADDR and DMAx_CURR_X_COUNT
- Control/status registers such as DMAx_IRQ_STATUS and DMAx_PERIPHERAL_MAP

All DMA registers can be accessed as 16-bit entities. However, the following registers may also be accessed as 32-bit registers.

- DMAx_NEXT_DESC_PTR
- DMAx_START_ADDR
- DMAx_CURR_DESC_PTR
- DMAx_CURR_ADDR

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses different naming conventions for physical registers and their corresponding elements in descriptors that reside in memory. Table 5-6 shows the relation.

DMA Peripheral Map Registers DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP)

Each DMA channel's DMAx_PERIPHERAL_MAP register contains bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel

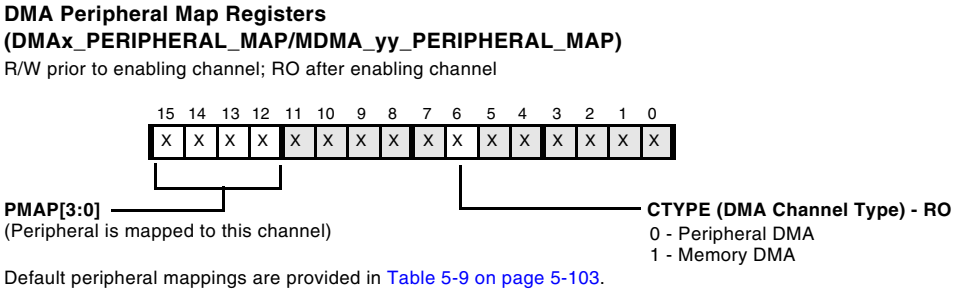

DMA Peripheral Map Registers
(DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
PMAP[3:0]
(Peripheral is mapped to this channel)
CTYPE (DMA Channel Type) - RO
0 - Peripheral DMA
1 - Memory DMA
Default peripheral mappings are provided in Table 5-9 on page 5-103.

Figure 5-5. DMA Peripheral Map Registers

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Make sure DMA is disabled on channels 6 and 7.
2. Write DMA6_PERIPHERAL_MAP with 0x7000 and DMA7_PERIPHERAL_MAP with 0x6000.
3. Enable DMA on channels 6 and/or 7.

DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)

The DMAx_CONFIG register, shown in Figure 5-6, is used to set up DMA parameters and operating modes. Writing the DMAx_CONFIG register while DMA is already running will cause a DMA error unless writing with the DMAEN bit set to 0.

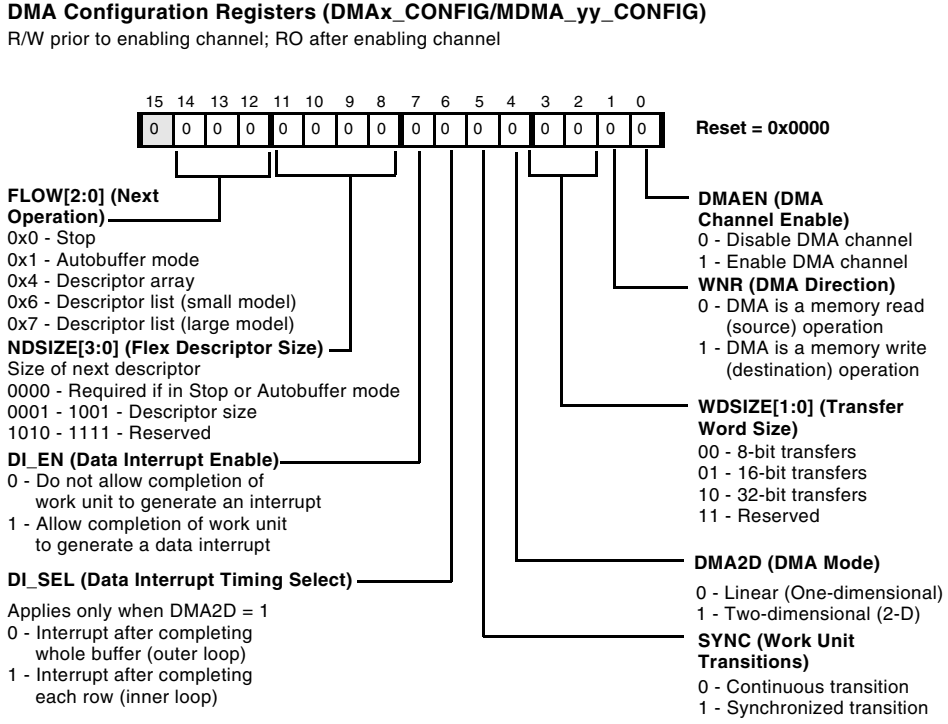

DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
FLOW[2:0] (Next Operation)
0x0 - Stop
0x1 - Autobuffer mode
0x4 - Descriptor array
0x6 - Descriptor list (small model)
0x7 - Descriptor list (large model)
NDSIZE[3:0] (Flex Descriptor Size)
Size of next descriptor
0000 - Required if in Stop or Autobuffer mode
0001 - 1001 - Descriptor size
1010 - 1111 - Reserved
DI_EN (Data Interrupt Enable)
0 - Do not allow completion of work unit to generate an interrupt
1 - Allow completion of work unit to generate a data interrupt
DI_SEL (Data Interrupt Timing Select)
Applies only when DMA2D = 1
0 - Interrupt after completing whole buffer (outer loop)
1 - Interrupt after completing each row (inner loop)
DMAEN (DMA Channel Enable)
0 - Disable DMA channel
1 - Enable DMA channel
WNR (DMA Direction)
0 - DMA is a memory read (source) operation
1 - DMA is a memory write (destination) operation
WDSIZE[1:0] (Transfer Word Size)
00 - 8-bit transfers
01 - 16-bit transfers
10 - 32-bit transfers
11 - Reserved
DMA2D (DMA Mode)
0 - Linear (One-dimensional)
1 - Two-dimensional (2-D)
SYNC (Work Unit Transitions)
0 - Continuous transition
1 - Synchronized transition

Figure 5-6. DMA Configuration Registers

The fields of the DMAx_CONFIG register are used to set up DMA parameters and operating modes.

- FLOW[2:0] (next operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:
- 0x0 - stop. When the current work unit completes, the DMA channel stops automatically, after signaling an interrupt (if selected). The DMA_RUN status bit in the DMAx_IRQ_STATUS register changes from 1 to 0, while the DMAEN bit in the DMAx_CONFIG register is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to the DMAx_CONFIG register specifying the next work unit, in which the DMAEN bit is set to 1.

  0x1 - autobuffer mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular buffer fashion based on user-programmed DMA MMR settings. Upon completion of the work unit, the parameter registers are reloaded into the current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMAEN bit in the DMAx_CONFIG register.

  0x4 - descriptor array mode. This mode fetches a descriptor from memory that does not include the NDPH or NDPL elements. Because the descriptor does not contain a next descriptor pointer entry, the DMA engine defaults to using the DMAx_CURR_DESC_PTR register to step through descriptors, thus allowing a group of descriptors to follow one another in memory like an array.

  0x6 - descriptor list (small model) mode. This mode fetches a descriptor from memory that includes NDPL, but not NDPH. Therefore, the high 16 bits of the next descriptor pointer field are taken from the upper 16 bits of the DMAx_NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.

0x7 - descriptor list (large model) mode. This mode fetches a descriptor from memory that includes NDPH and NDPL, thus allowing maximum flexibility in locating descriptors in memory.

- NDSIZE[3:0] (flex descriptor size). This field specifies the number of descriptor elements in memory to load. This field must be 0 if in stop or autobuffer mode. If NDSIZE and FLOW specify a descriptor that extends beyond YMOD, a DMA error results.
- DI_EN (data interrupt enable). This bit specifies whether to allow completion of a work unit to generate a data interrupt.
- DI_SEL (data interrupt timing select). This bit specifies the timing of a data interrupt—after completing the whole buffer or after completing each row of the inner loop. This bit is used only in 2-D DMA operation.
- SYNC (work unit transitions). This bit specifies whether the DMA channel performs a continuous transition (SYNC = 0) or a synchronized transition (SYNC = 1) between work units. For more information, see "Work Unit Transitions" on page 5-24.

  In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work unit.

Work unit transitions for MDMA streams are controlled by the SYNC bit of the MDMA source channel's DMAx_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be set to 0.

- DMA2D (DMA mode). This bit specifies whether DMA mode involves only DMAx_X_COUNT and DMAx_X_MODIFY (one-dimensional DMA) or also involves DMAx_Y_COUNT and DMAx_Y_MODIFY (two-dimensional DMA).
- WDSIZE[1:0] (transfer word size). The DMA engine supports transfers of 8-, 16-, or 32-bit items. Each request/grant results in a single memory access (although two cycles are required to transfer 32-bit data through a 16-bit memory port or through the 16-bit DMA access bus). The increment sizes (strides) of the DMA address pointer registers must be a multiple of the transfer unit size—one for 8-bit, two for 16-bit, four for 32-bit.

  Only SPORT DMA and Memory DMA can operate with a transfer size of 32 bits. All other peripherals have a maximum DMA transfer size of 16 bits.
- WNR (DMA direction). This bit specifies DMA direction—memory read (0) or memory write (1).
- DMAEN (DMA channel enable). This bit specifies whether to enable a given DMA channel.

When a peripheral DMA channel is enabled, interrupts from the peripheral denote DMA requests. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller. To avoid unexpected results, take care to enable the DMA channel before enabling the peripheral, and to disable the peripheral before disabling the DMA channel.

DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

The DMAx_IRQ_STATUS register, shown in Figure 5-7, contains bits that record whether the DMA channel:

- Is enabled and operating, enabled but stopped, or disabled.
- Is fetching data or a DMA descriptor.
- Has detected that a global DMA interrupt or a channel interrupt is being asserted.
- Has logged occurrence of a DMA error.

Note the DMA_DONE interrupt is asserted when the last memory access (read or write) has completed.

For a memory transfer to a peripheral, there may be up to four data words in the channel's DMA FIFO when the interrupt occurs. At this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the DMA_RUN bit. As long as there is undelivered transmit data in the FIFO, the DMA_RUN bit is 1.

For a memory write DMA channel, the state of the DMA_RUN bit has no meaning after the last DMA_DONE event has been signaled. It does not indicate the status of the DMA FIFO.

For MDMA transfers where an interrupt is not desired to notify when the DMA operation has ended, software should poll the DMA_DONE bit, rather than the DMA_RUN bit to determine when the transaction has completed.

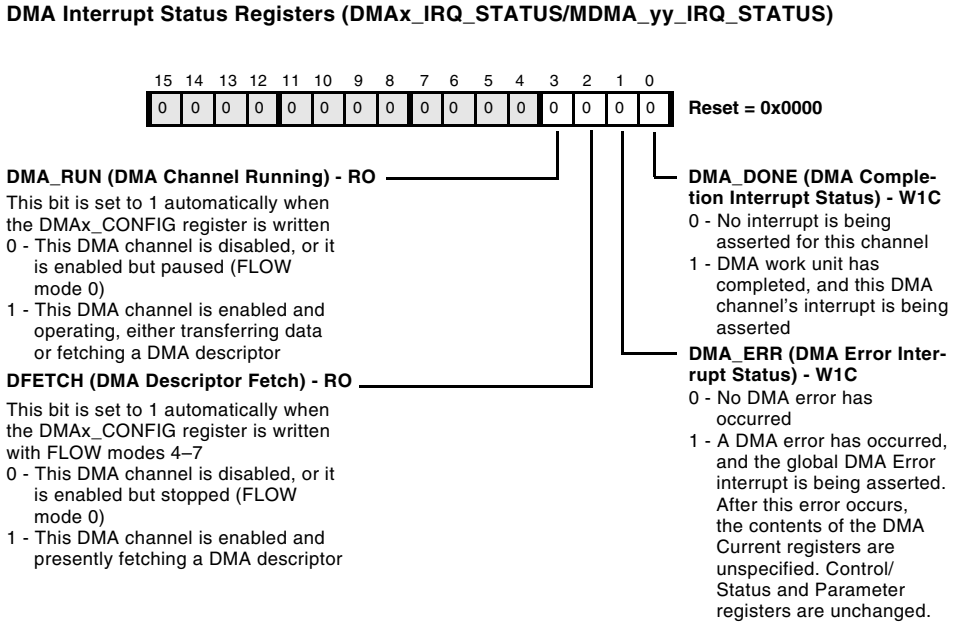

DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DMA_RUN (DMA Channel Running) - RO
This bit is set to 1 automatically when the DMAx_CONFIG register is written
0 - This DMA channel is disabled, or it is enabled but paused (FLOW mode 0)
1 - This DMA channel is enabled and operating, either transferring data or fetching a DMA descriptor
DFETCH (DMA Descriptor Fetch) - RO
This bit is set to 1 automatically when the DMAx_CONFIG register is written with FLOW modes 4–7
0 - This DMA channel is disabled, or it is enabled but stopped (FLOW mode 0)
1 - This DMA channel is enabled and presently fetching a DMA descriptor
DMA_DONE (DMA Completion Interrupt Status) - W1C
0 - No interrupt is being asserted for this channel
1 - DMA work unit has completed, and this DMA channel's interrupt is being asserted
DMA_ERR (DMA Error Interrupt Status) - W1C
0 - No DMA error has occurred
1 - A DMA error has occurred, and the global DMA Error interrupt is being asserted. After this error occurs, the contents of the DMA Current registers are unspecified. Control/ Status and Parameter registers are unchanged.

Figure 5-7. DMA Interrupt Status Registers

The processor supports a flexible interrupt control structure with three interrupt sources:

- Data driven interrupts (see Table 5-7)
- Peripheral error interrupts
- DMA error interrupts (for example, bad descriptor or bus error)

Separate interrupt request (IRQ) levels are allocated for data, peripheral error, and DMA error interrupts.

The DMA error conditions for all DMA channels are OR'd together into one system-level DMA error interrupt. The individual `IRQ_STATUS` words of each channel can be read to identify the channel that caused the DMA error interrupt.

Table 5-7. Data Driven Interrupts

Interrupt Name	Description
No Interrupt	Interrupts can be disabled for a given work unit.
Peripheral Interrupt	These are peripheral (non-DMA) interrupts.
Row Completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer Completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

Note the DMA_DONE and DMA_ERR interrupt indicators are write-one-to-clear (W1C).

When switching a peripheral from DMA to non-DMA mode, the peripheral's interrupts should be disabled during the mode switch (via the appropriate peripheral register or SIC_IMASK register) so that no unintended interrupt is generated on the shared DMA/interrupt request line.

DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR)

The DMAx_START_ADDR register, shown in Figure 5-8, contains the start address of the data buffer currently targeted for DMA.

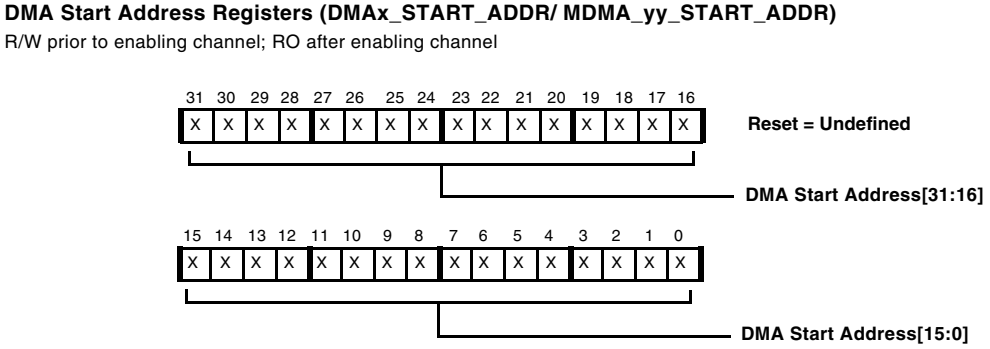

DMA Start Address Registers (DMAx_START_ADDR/ MDMA_yy_START_ADDR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
DMA Start Address[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
DMA Start Address[15:0]

Figure 5-8. DMA Start Address Registers

DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)

The 32-bit DMAx_CURR_ADDR register shown in Figure 5-9, contains the present DMA transfer address for a given DMA session. On the first memory transfer of a DMA work unit, the DMAx_CURR_ADDR register is loaded from the DMAx_START_ADDR register, and it is incremented as each transfer occurs.

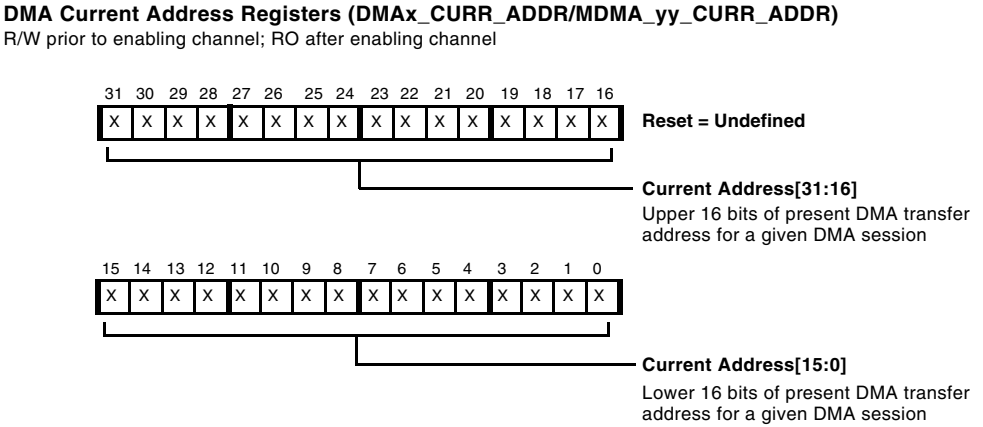

DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Current Address[31:16]
Upper 16 bits of present DMA transfer address for a given DMA session
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Current Address[15:0]
Lower 16 bits of present DMA transfer address for a given DMA session

Figure 5-9. DMA Current Address Registers

DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)

For 2-D DMA, the DMAx_X_COUNT register, shown in Figure 5-10, contains the inner loop count. For 1-D DMA, it specifies the number of elements to transfer. For details, see "Two-Dimensional DMA Operation" on page 5-11. A value of 0 in DMAx_X_COUNT corresponds to 65,536 elements.

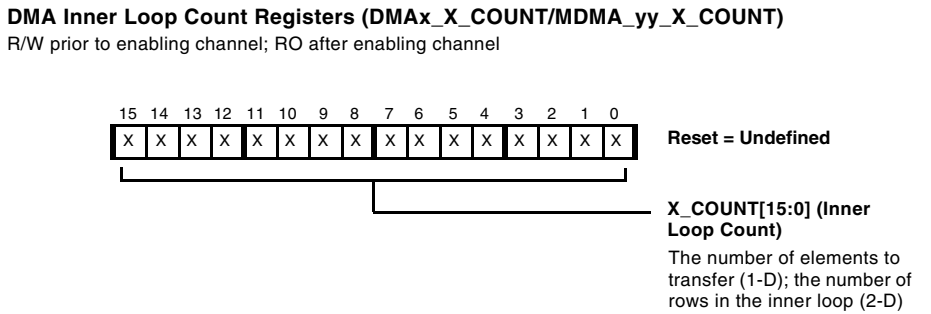

DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
X_COUNT[15:0] (Inner Loop Count)
The number of elements to transfer (1-D); the number of rows in the inner loop (2-D)

Figure 5-10. DMA Inner Loop Count Registers

DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT)

The DMAx_CURR_X_COUNT register, shown in Figure 5-11, holds the number of transfers remaining in the current DMA row (inner loop). On the first memory transfer of each DMA work unit, it is loaded with the value in the DMAx_X_COUNT register and then decremented. For 2-D DMA, on the last memory transfer in each row except the last row, it is reloaded with the value in the DMAx_X_COUNT register; this occurs at the same time that the value in the DMAx_CURR_Y_COUNT register is decremented. Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete. In 2-D DMA, the DMAx_CURR_X_COUNT register value is 0 only when the entire transfer is complete. Between rows it is equal to the value of the DMAx_X_COUNT register.

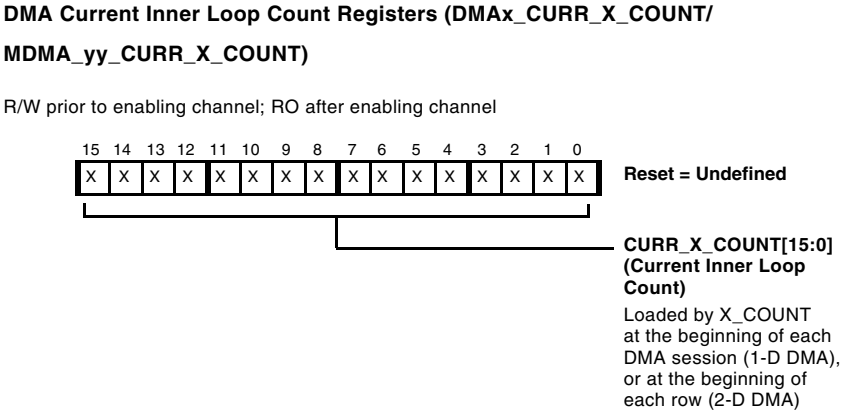

DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT/
MDMA_yy_CURR_X_COUNT)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
CURR_X_COUNT[15:0]
(Current Inner Loop
Count)
Loaded by X_COUNT
at the beginning of each
DMA session (1-D DMA),
or at the beginning of
each row (2-D DMA)

Figure 5-11. DMA Current Inner Loop Count Registers

DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)

The DMAx_X_MODIFY register, shown in Figure 5-12, contains a signed, two's-complement byte-address increment. In 1-D DMA, this increment is the stride that is applied after transferring each element.

DMAx_X_MODIFY is specified in bytes, regardless of the DMA transfer size.

In 2-D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, the DMAx_Y_MODIFY register is applied instead, except on the very last transfer of each work unit. The DMAx_X_MODIFY register is always applied to the last transfer of a work unit.

The DMAx_X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in transferring data between a data register and an external memory-mapped peripheral.

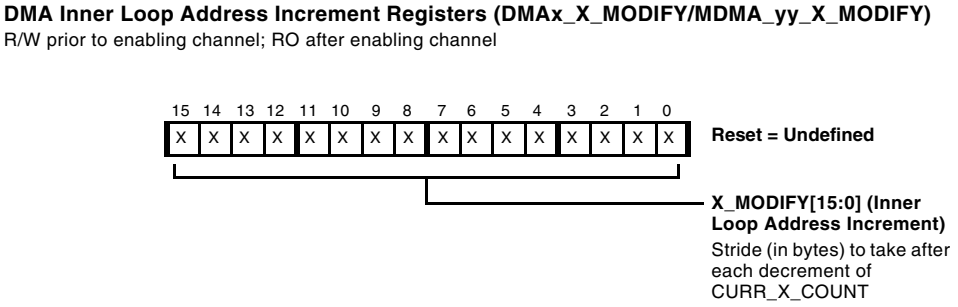

DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
X_MODIFY[15:0] (Inner Loop Address Increment)
Stride (in bytes) to take after each decrement of CURR_X_COUNT

Figure 5-12. DMA Inner Loop Address Increment Registers

DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)

For 2-D DMA, the DMAx_Y_COUNT register, shown in Figure 5-13, contains the outer loop count. It is not used in 1-D DMA mode. This register contains the number of rows in the outer loop of a 2-D DMA sequence. For details, see "Two-Dimensional DMA Operation" on page 5-11.

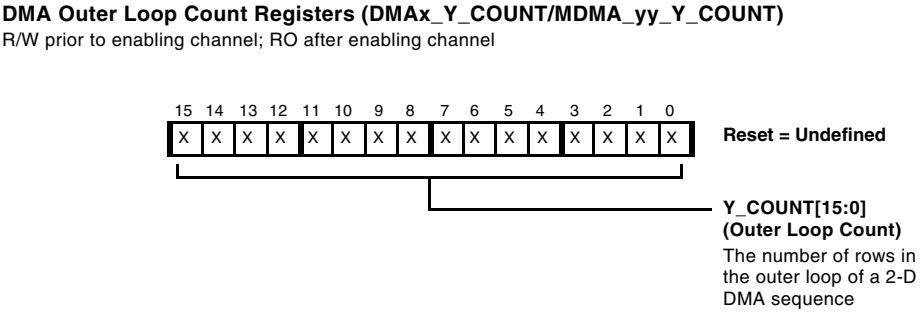

DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)
R/W prior to enabling channel; RO after enabling channel
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
Y_COUNT[15:0]
(Outer Loop Count)
The number of rows in the outer loop of a 2-D DMA sequence

Figure 5-13. DMA Outer Loop Count Registers

DMA Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/ MDMA_yy_CURR_Y_COUNT)

The DMAx_CURR_Y_COUNT register, used only in 2-D mode, holds the number of full or partial rows (outer loops) remaining in the current work unit. See Figure 5-14. On the first memory transfer of each DMA work unit, it is loaded with the value of the DMAx_Y_COUNT register. The register is decremented each time the DMAx_CURR_X_COUNT register expires during 2-D DMA operation (1 to DMAx_X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2-D DMA session is complete, DMAx_CURR_Y_COUNT = 1 and DMAx_CURR_X_COUNT = 0.

DMA Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT)

R/W prior to enabling channel; RO after enabling channel

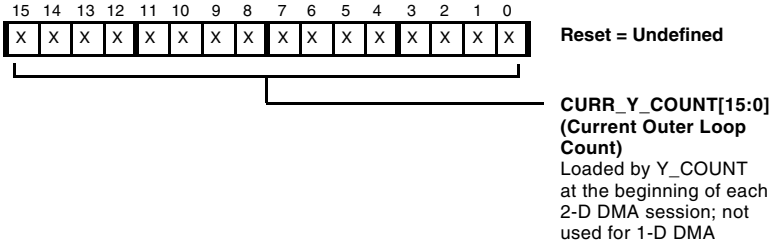

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
CURR_Y_COUNT[15:0] (Current Outer Loop Count)
Loaded by Y_COUNT at the beginning of each 2-D DMA session; not used for 1-D DMA

Figure 5-14. DMA Current Outer Loop Count Registers

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY)

The DMAx_Y_MODIFY register contains a signed, two's-complement value. See Figure 5-15. This byte-address increment is applied after each decrement of the DMAx_CURR_Y_COUNT register except for the last item in the 2-D array where the DMAx_CURR_Y_COUNT also expires. The value is the offset between the last word of one row and the first word of the next row. For details, see "Two-Dimensional DMA Operation" on page 5-11.

DMAx_Y_MODIFY is specified in bytes, regardless of the DMA transfer size.

DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/ MDMA_yy_Y_MODIFY)

R/W prior to enabling channel; RO after enabling channel

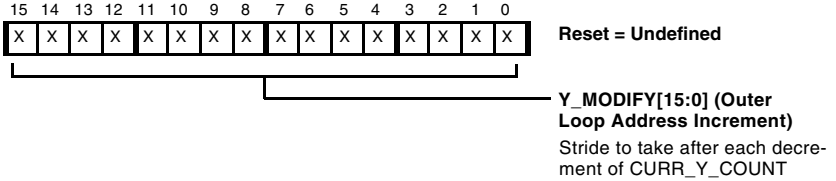

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Reset = Undefined
Y_MODIFY[15:0] (Outer Loop Address Increment)
Stride to take after each decrement of CURR_Y_COUNT

Figure 5-15. DMA Outer Loop Address Increment Registers

DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR)

The 32-bit DMAx_NEXT_DESC_PTR register, shown in Figure 5-16, specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, this register is copied into the DMAx_CURR_DESC_PTR register. Then, during the descriptor fetch, the DMAx_CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, the DMAx_NEXT_DESC_PTR register, and not the DMAx_CURR_DESC_PTR register, must be programmed directly via MMR access before starting DMA operation.

In descriptor array mode, the next descriptor pointer register is disregarded, and fetching is controlled only by the DMAx_CURR_DESC_PTR register.

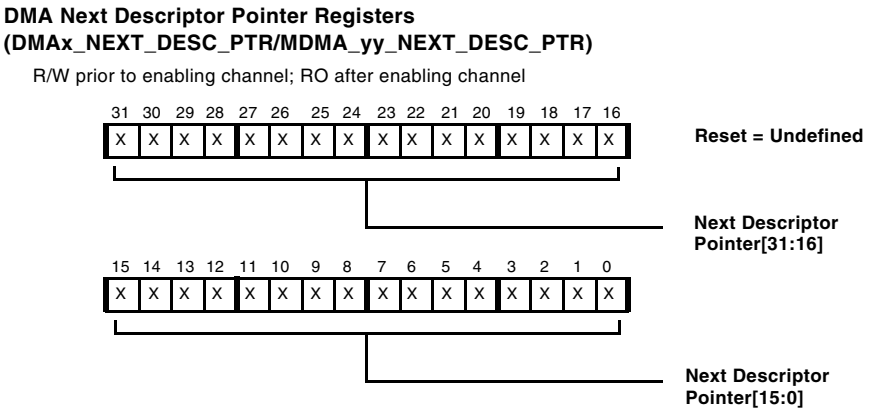

DMA Next Descriptor Pointer Registers
(DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Next Descriptor Pointer[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Next Descriptor Pointer[15:0]

Figure 5-16. DMA Next Descriptor Pointer Registers

DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)

The 32-bit DMAx_CURR_DESC_PTR register, shown in Figure 5-17, contains the memory address for the next descriptor element to be loaded. For FLOW mode settings that involve descriptors (FLOW = 4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For descriptor list modes (FLOW = 6 or 7), this register is initialized from the DMAx_NEXT_DESC_PTR register before loading each descriptor. Then, the address in the DMAx_CURR_DESC_PTR register increments as each descriptor element is read in.

When the entire descriptor has been read, the DMAx_CURR_DESC_PTR register contains this value:

Descriptor Start Address + (2 × Descriptor Size) (# of elements)

For descriptor array mode (FLOW = 4), this register, and not the DMAx_NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

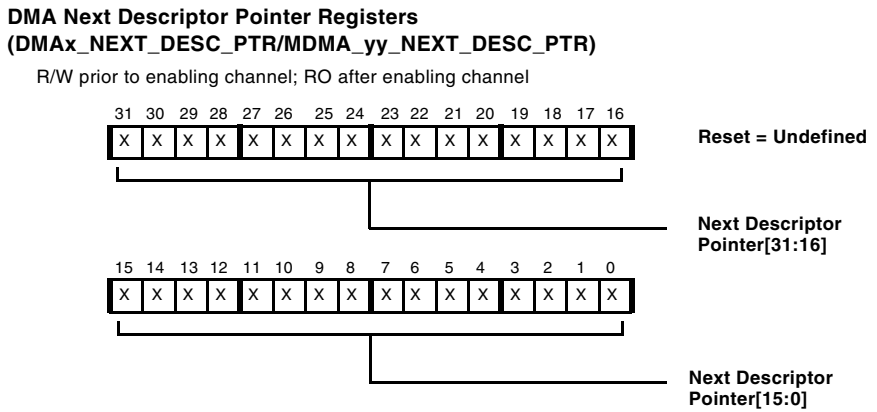

DMA Next Descriptor Pointer Registers
(DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR)
R/W prior to enabling channel; RO after enabling channel
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Next Descriptor Pointer[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Next Descriptor Pointer[15:0]

Figure 5-17. DMA Current Descriptor Pointer Registers

HMDMA Registers

Some processors have two HMDMA blocks, while others have none. See the "Unique Information for the ADSP-BF59x Processor" on page 5-101 to determine whether this feature is applicable to your product. HMDMA0 is associated with MDMA0, and HMDMA1 is associated with MDMA1.

Handshake MDMA Control Registers (HMDMAx_CONTROL)

The HMDMAx_CONTROL register, shown in Figure 5-18, is used to set up HMDMA parameters and operating modes.

The DRQ[1:0] field is used to control the priority of the MDMA channel when the HMDMA is disabled, that is, when handshake control is not being used (see Table 5-8).

Table 5-8. DRQ[1:0] Values

DRQ[1:0]	Priority	Description
00	Disabled	The MDMA request is disabled.
01	Enabled/S	Normal MDMA channel priority. The channel in this mode is limited to single memory transfers separated by one idle system clock. Request single transfer from MDMA channel.
10	Enabled/M	Normal MDMA channel functionality and priority. Request multiple transfers from MDMA channel (default).
11	Urgent	The MDMA channel priority is elevated to urgent. In this state, it has higher priority for memory access than non-urgent channels. If two channels are both urgent, the lower-numbered channel has priority.

The RBC bit forces the BCOUNT register to be reloaded with the BCINIT value while the module is already active. Do not set this bit in the same write that sets the HMDMAEN bit to active.

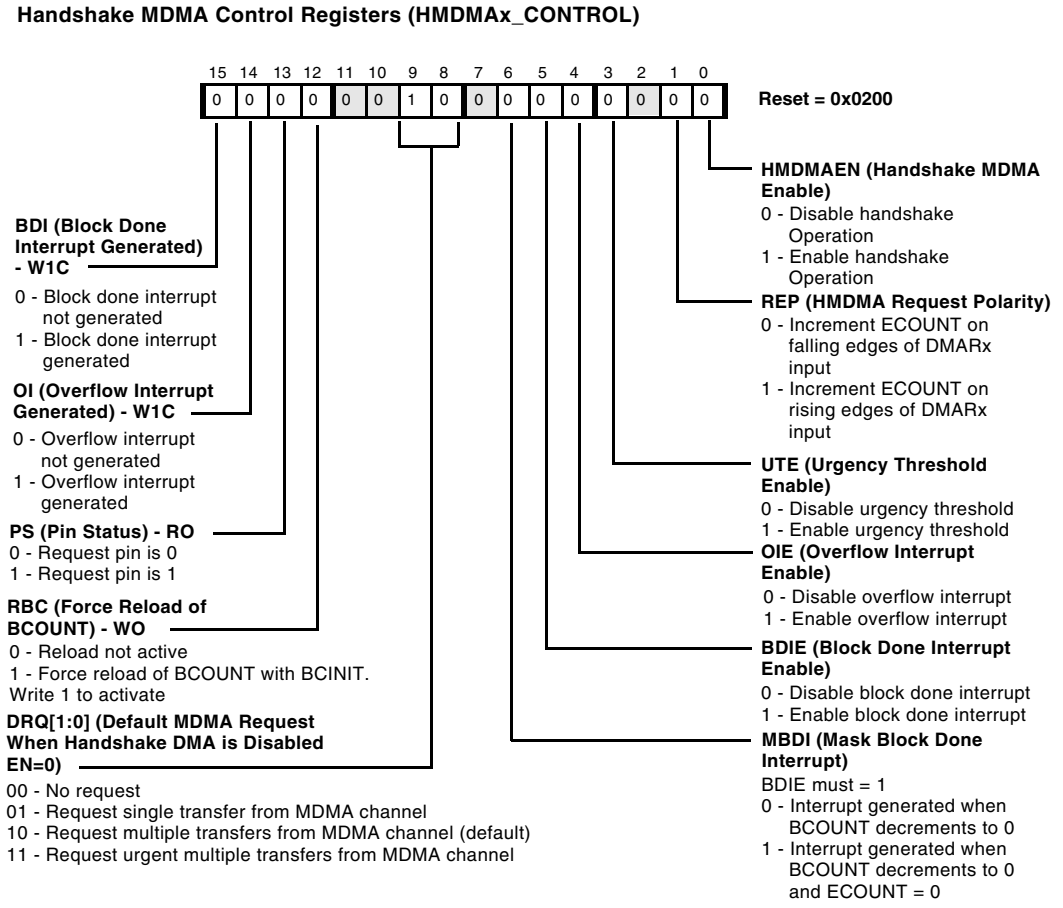

Handshake MDMA Control Registers (HMDMAx_CONTROL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Reset = 0x0200
HMDMAEN (Handshake MDMA Enable)
0 - Disable handshake Operation
1 - Enable handshake Operation
REP (HMDMA Request Polarity)
0 - Increment ECOUNT on falling edges of DMARx input
1 - Increment ECOUNT on rising edges of DMARx input
UTE (Urgency Threshold Enable)
0 - Disable urgency threshold
1 - Enable urgency threshold
OIE (Overflow Interrupt Enable)
0 - Disable overflow interrupt
1 - Enable overflow interrupt
BDIE (Block Done Interrupt Enable)
0 - Disable block done interrupt
1 - Enable block done interrupt
MBDI (Mask Block Done Interrupt)
BDIE must = 1
0 - Interrupt generated when BCOUNT decrements to 0
1 - Interrupt generated when BCOUNT decrements to 0 and ECOUNT = 0
BDI (Block Done Interrupt Generated) - W1C
0 - Block done interrupt not generated
1 - Block done interrupt generated
OI (Overflow Interrupt Generated) - W1C
0 - Overflow interrupt not generated
1 - Overflow interrupt generated
PS (Pin Status) - RO
0 - Request pin is 0
1 - Request pin is 1
RBC (Force Reload of BCOUNT) - WO
0 - Reload not active
1 - Force reload of BCOUNT with BCINIT. Write 1 to activate
DRQ[1:0] (Default MDMA Request When Handshake DMA is Disabled EN=0)
00 - No request
01 - Request single transfer from MDMA channel
10 - Request multiple transfers from MDMA channel (default)
11 - Request urgent multiple transfers from MDMA channel

Figure 5-18. Handshake MDMA Control Registers

Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT)

The HMDMAx_BCINIT register, shown in Figure 5-19, holds the number of transfers to do per edge of the DMARx control signal.

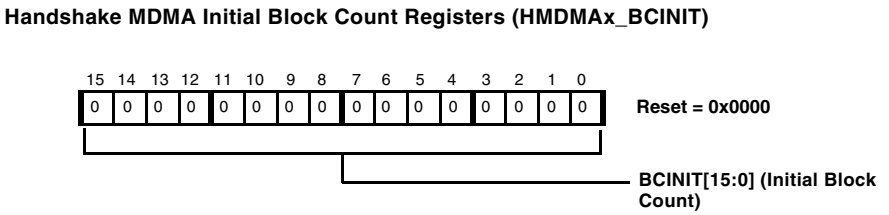

Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
BCINIT[15:0] (Initial Block Count)

Figure 5-19. Handshake MDMA Initial Block Count Registers

Handshake MDMA Current Block Count Registers (HMDMAx_BCOUNT)

The HMDMAx_BCOUNT register, shown in Figure 5-20, holds the number of transfers remaining for the current edge. MDMA requests are generated if this count is greater than 0.

Examples:

- 0000 = 0 transfers remaining
- FFFF = 65535 transfers remaining

The BCOUNT field is loaded with BCINIT when ECOUNT is greater than 0 and BCOUNT is expired (0). Also, if the RBC bit in the HMDMAx_CONTROL register is written to 1, BCOUNT is loaded with BCINIT. The BCOUNT field is decremented with each MDMA grant. It is cleared when HMDMA is disabled.

A block done interrupt is generated when BCOUNT decrements to 0. If the MBDI bit in the HMDMAx_CONTROL register is set, the interrupt is suppressed until ECOUNT is 0. If BCINIT is 0, no block done interrupt is generated, since no DMA requests were generated or grants received.

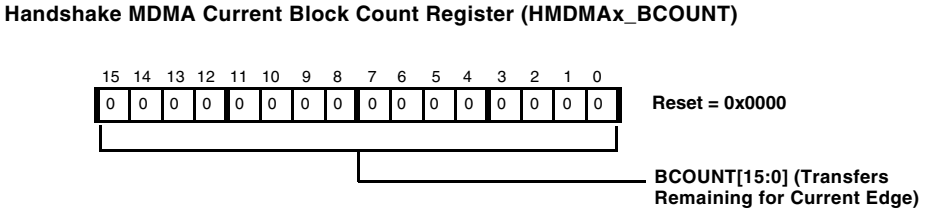

Handshake MDMA Current Block Count Register (HMDMAx_BCOUNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
BCOUNT[15:0] (Transfers Remaining for Current Edge)

Figure 5-20. Handshake MDMA Current Block Count Registers

Handshake MDMA Current Edge Count Registers (HMDMAx_ECOUNT)

The HMDMAx_ECOUNT register, shown in Figure 5-21, holds a signed number of edges remaining to be serviced. This number is in a signed two's complement representation. When an edge is detected on the respective DMARx input, requests occur if this count is greater than or equal to 0 and BCOUNT is greater than 0.

When the handshake mode is enabled, ECOUNT is loaded and the resulting number of requests is:

Number of edges + N,

where N is the number loaded from ECINIT. The number N can be positive or negative. Examples:

- 0x7FFF = 32,767 edges remaining
- 0x0000 = 0 edges remaining
- 0x8000 = –32,768: ignore the next 32,768 edges

Each time that BCOUNT expires, ECOUNT is decremented and BCOUNT is reloaded from BCINIT. When a handshake request edge is detected, ECOUNT is incremented. The ECOUNT field is cleared when HMDMA is disabled.

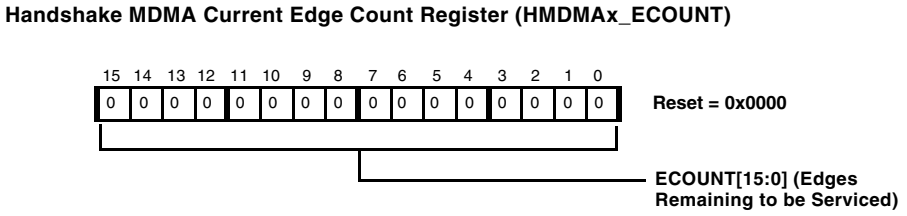

Handshake MDMA Current Edge Count Register (HMDMAx_ECOUNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ECOUNT[15:0] (Edges Remaining to be Serviced)

Figure 5-21. Handshake MDMA Current Edge Count Registers

Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT)

The HMDMAx_ECINIT register, shown in Figure 5-22, holds a signed number that is loaded into HMDMAx_ECOUNT when handshake DMA is enabled. This number is in a signed two's complement representation.

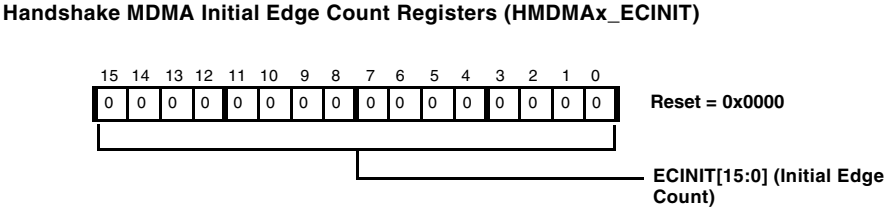

Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ECINIT[15:0] (Initial Edge Count)

Figure 5-22. Handshake MDMA Initial Edge Count Registers

Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT)

The HMDMAx_ECURGENT register, shown in Figure 5-23, holds the urgent threshold. If the ECOUNT field in the HMDMAx_ECOUNT register is greater than this threshold, the MDMA request is urgent and might get higher priority.

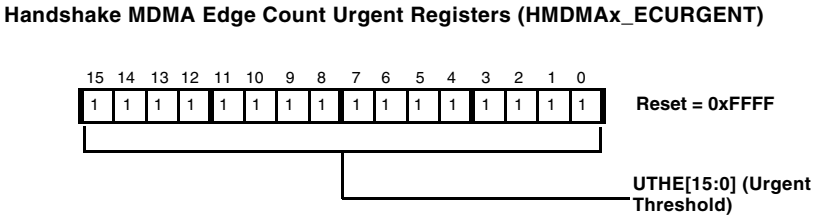

Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Reset = 0xFFFF
UTHE[15:0] (Urgent Threshold)

Figure 5-23. Handshake MDMA Edge Count Urgent Registers

Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW)

The HMDMAx_ECOVERFLOW register, shown in Figure 5-24, holds the interrupt threshold. If the ECOUNT field in the HMDMAx_ECOUNT register is greater than this threshold, an overflow interrupt is generated.

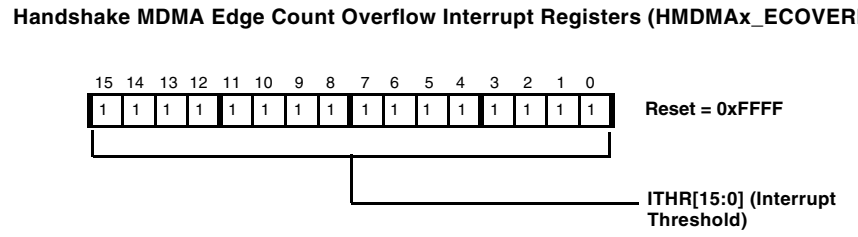

Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Reset = 0xFFFF
ITHR[15:0] (Interrupt Threshold)

Figure 5-24. Handshake MDMA Edge Count Overflow Interrupt Registers

DMA Traffic Control Registers (DMA_TC_PER and DMA_TC_CNT)

The DMA_TC_PER register (see Figure 5-25) and the DMA_TC_CNT register (see Figure 5-26) work with other DMA registers to define traffic control.

DMA_TC_PER Register

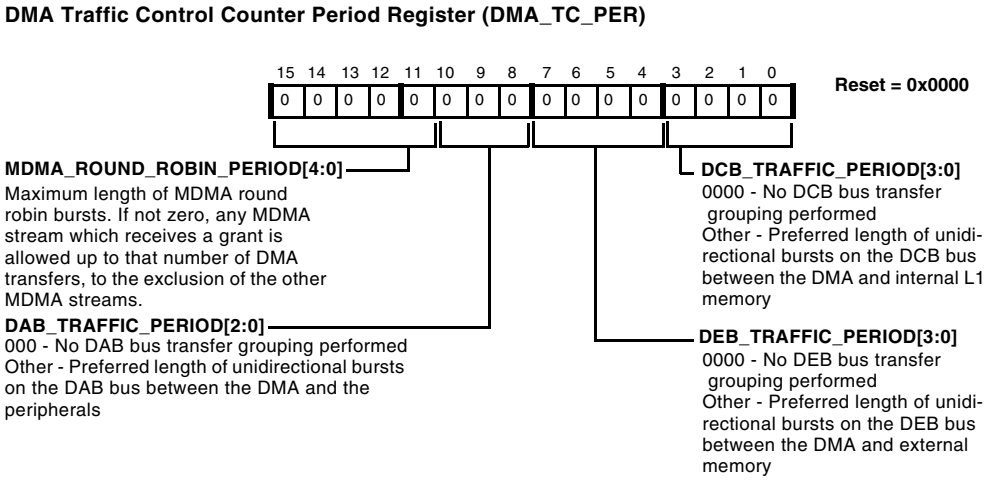

DMA Traffic Control Counter Period Register (DMA_TC_PER)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MDMA_ROUND_ROBIN_PERIOD[4:0]
Maximum length of MDMA round robin bursts. If not zero, any MDMA stream which receives a grant is allowed up to that number of DMA transfers, to the exclusion of the other MDMA streams.
DAB_TRAFFIC_PERIOD[2:0]
000 - No DAB bus transfer grouping performed
Other - Preferred length of unidirectional bursts on the DAB bus between the DMA and the peripherals
DCB_TRAFFIC_PERIOD[3:0]
0000 - No DCB bus transfer grouping performed
Other - Preferred length of unidirectional bursts on the DCB bus between the DMA and internal L1 memory
DEB_TRAFFIC_PERIOD[3:0]
0000 - No DEB bus transfer grouping performed
Other - Preferred length of unidirectional bursts on the DEB bus between the DMA and external memory

Figure 5-25. DMA Traffic Control Counter Period Register

DMA_TC_CNT Register

The MDMA_ROUND_ROBIN_COUNT field shows the current transfer count remaining in the MDMA round-robin period. It initializes to MDMA_ROUND_ROBIN_PERIOD whenever DMA_TC_PER is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle. It then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

DMA Traffic Control Counter Register (DMA_TC_CNT)
RO

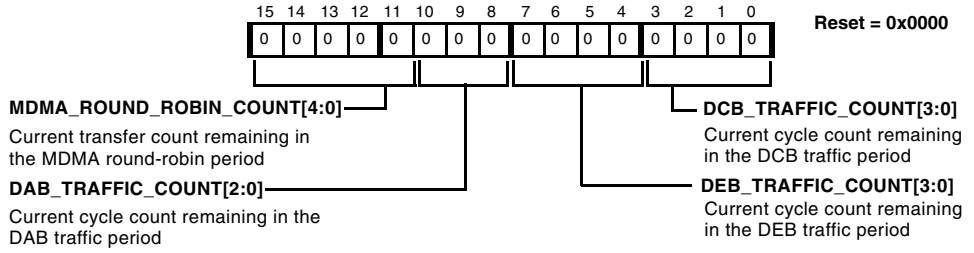

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MDMA_ROUND_ROBIN_COUNT[4:0]
Current transfer count remaining in the MDMA round-robin period
DAB_TRAFFIC_COUNT[2:0]
Current cycle count remaining in the DAB traffic period
DCB_TRAFFIC_COUNT[3:0]
Current cycle count remaining in the DCB traffic period
DEB_TRAFFIC_COUNT[3:0]
Current cycle count remaining in the DEB traffic period

Figure 5-26. DMA Traffic Control Counter Register

The DAB_TRAFFIC_COUNT field shows the current cycle count remaining in the DAB traffic period. It initializes to DAB_TRAFFIC_PERIOD whenever DMA_TC_PER is written, or whenever the DAB bus changes direction or becomes idle. It then counts down from DAB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DAB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DAB access is treated preferentially, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from DAB_TRAFFIC_PERIOD to begin a new burst.

The DEB_TRAFFIC_COUNT field shows the current cycle count remaining in the DEB traffic period. It initializes to DEB_TRAFFIC_PERIOD whenever DMA_TC_PER is written or whenever the DEB bus changes direction or becomes idle. It then counts down from DEB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DEB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DEB access is treated preferentially, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from DEB_TRAFFIC_PERIOD to begin a new burst.

The DCB_TRAFFIC_COUNT field shows the current cycle count remaining in the DCB traffic period. It initializes to DCB_TRAFFIC_PERIOD whenever DMA_TC_PER is written or whenever the DCB bus changes direction or becomes idle. It then counts down from DCB_TRAFFIC_PERIOD to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DCB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DCB access is treated preferentially, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from DCB_TRAFFIC_PERIOD to begin a new burst.

Programming Examples

The following examples illustrate memory DMA and handshaked memory DMA basics. Examples for peripheral DMAs can be found in the respective peripheral chapters.

Register-Based 2-D Memory DMA

Listing 5-1 shows a register-based, two-dimensional MDMA. While the source channel processes linearly, the destination channel re-sorts elements by transposing the two-dimensional data array. See Figure 5-27.

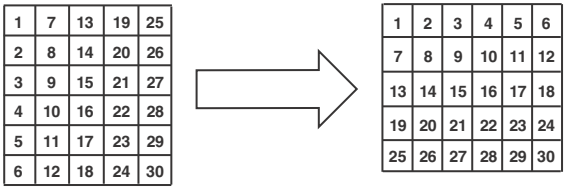


Figure 5-27. DMA Example, 2-D Array

The two arrays reside in two different L1 data memory blocks. However, the arrays could reside in any internal or external memory, including L1

instruction memory and SDRAM. For the case where the destination array resided in SDRAM, it is a good idea to let the source channel re-sort elements and to let the destination buffer store linearly.

Listing 5-1. Register-Based 2-D Memory DMA

```
#include <defBF527.h>/*For ADSP-BF527 product, as an example.*/
#define X 5
#define Y 6

.section L1_data_a;
.byte2 aSource[X*Y] =
    1,  7, 13, 19, 25,
    2,  8, 14, 20, 26,
    3,  9, 15, 21, 27,
    4, 10, 16, 22, 28,
    5, 11, 17, 23, 29,
    6, 12, 18, 24, 30;

.section L1_data_b;
.byte2 aDestination[X*Y];

.section L1_code;
.global _main;
_main:
    p0.l = lo(MDMA_S0_CONFIG);
    p0.h = hi(MDMA_S0_CONFIG);
    call memdma_setup;
    call memdma_wait;
_main.forever:
    jump _main.forever;
_main.end:
```

The setup routine shown in Listing 5-2 initializes either MDMA0 or MDMA1, depending on whether the MMR address of `MDMA_S0_CONFIG` or

MDMA_S1_CONFIG is passed in the P0 register. Note that the source channel is enabled before the destination channel. Also, it is common to synchronize interrupts with the destination channel because only those interrupts indicate completion of both DMA read and write operations.

Listing 5-2. Two-Dimensional Memory DMA Setup Example

```
memdma_setup:
    [--sp] = r7;
/* setup 1D source DMA for 16-bit transfers */
    r7.l = lo(aSource);
    r7.h = hi(aSource);
    [p0 + MDMA_S0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2;
    w[p0 + MDMA_S0_X_MODIFY - MDMA_S0_CONFIG] = r7;
     r7.l = X * Y;
    w[p0 + MDMA_S0_X_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = WDSIZE_16 | DMAEN;
    w[p0] = r7;
/* setup 2D destination DMA for 16-bit transfers */
    r7.l = lo(aDestination);
    r7.h = hi(aDestination);
    [p0 + MDMA_D0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2*Y;
    w[p0 + MDMA_D0_X_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = Y;
    w[p0 + MDMA_D0_Y_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = X;
    w[p0 + MDMA_D0_X_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = -2 * (Y * (X-1) - 1);
    w[p0 + MDMA_D0_Y_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = DMA2D | DI_EN | WDSIZE_16 | WNR | DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
```

```
    rts;
memdma_setup.end:
```

For simplicity the example shown in Listing 5-3 polls the DMA status rather than using interrupts, which is the normal case in a real application.

Listing 5-3. Polling DMA Status

```
memdma_wait:
    [--sp] = r7;
memdma_wait.test:
    r7 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r7, bitpos(DMA_DONE));
    if !CC jump memdma_wait.test;
    r7 = DMA_DONE (z);
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_wait.end:
```

Initializing Descriptors in Memory

Descriptor-based DMAs expect the descriptor data to be available in memory by the time the DMA is enabled. Often, the descriptors are programmed by software at run-time. Many times, however, the descriptors—or at least large portions of them—can be static and therefore initialized at boot time. How to set up descriptors in global memory depends heavily on the programming language and the tool set used. The following examples show how this is best performed in the VisualDSP++ tools' assembly language.

Listing 5-4 uses multiple variables of either 16-bit or 32-bit size to describe DMA descriptors. This example has two descriptors in small list flow mode that point to each other. At the end of the second work unit, an interrupt is generated without discontinuing the DMA processing. The

trailing `.end` label is required to let the linker know that a descriptor forms a logical unit. It prevents the linker from removing variables when optimizing.

Listing 5-4. Two Descriptors in Small List Flow Mode

```
.section sdram;
.byte2 arrBlock1[0x400];
.byte2 arrBlock2[0x800];

.section L1_data_a;
.byte2 descBlock1 = lo(descBlock2);
.var   descBlock1.addr = arrBlock1;
.byte2 descBlock1.cfg = FLOW_SMALL|NDSIZE_5|WDSIZE_16|DMAEN;
.byte2 descBlock1.len = length(arrBlock1);
       descBlock1.end:

.byte2 descBlock2 = lo(descBlock1);
.var   descBlock2.addr = arrBlock2;
.byte2 descBlock2.cfg =
FLOW_SMALL|NDSIZE_5|DI_EN|WDSIZE_16|DMAEN;
.byte2 descBlock2.len = length(arrBlock2);
       descBlock2.end:
```

Another method featured by the VisualDSP++ tools takes advantage of C-style structures in global header files. The header file `descriptors.h` could look like Listing 5-5.

Listing 5-5. Header File to Define Descriptor Structures

```
#ifndef __INCLUDE_DESCRIPTORS__
#define __INCLUDE_DESCRIPTORS__
#ifdef _LANGUAGE_C
typedef struct {
```

```
    void  *pStart;
    short dConfig;
    short dXCount;
    short dXModify;
    short dYCount;
    short dYModify;
} dma_desc_arr;

typedef struct {
    void  *pNext;
    void  *pStart;
    short dConfig;
    short dXCount;
    short dXModify;
    short dYCount;
    short dYModify;
} dma_desc_list;

#endif // _LANGUAGE_C
#endif // __INCLUDE_DESCRIPTORS__
```

Note that near pointers are not natively supported by the C language and, thus, pointers are always 32 bits wide. Therefore, the scheme above cannot be used directly for small list mode without giving up pointer syntax. The variable definition file is required to import the C-style header file and can finally take advantage of the structures. See Listing 5-6.

Listing 5-6. Using Descriptor Structures

```
#include "descriptors.h"
.import "descriptors.h";

.section L1_data_a;
.align 4;
.var arrBlock3[N];
```

```
.var arrBlock4[N];

.struct dma_desc_list descBlock3 = {
    descBlock4, arrBlock3,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_32 | DMAEN,
    length(arrBlock3), 4,
    0, 0        /* unused values */
};

.struct dma_desc_list descBlock4 = {
    descBlock3, arrBlock4,
    FLOW_LARGE | NDSIZE_7 | DI_EN | WDSIZE_32 | DMAEN,
    length(arrBlock4), 4,
    0, 0        /* unused values */
};
```

Software-Triggered Descriptor Fetch Example

Listing 5-7 demonstrates a large list of descriptors that provide `FLOW` = 0 (stop mode) configuration. Consequently, the DMA stops by itself as soon as the work unit has finished. Software triggers the next work unit by simply writing the proper value into the DMA configuration registers. Since these values instruct the DMA controller to fetch descriptors in large list mode, the DMA immediately fetches the descriptor, thus overwriting the configuration value again with the new settings when it is started.

Note the requirement that source and destination channels stop after the same number of transfers. Between stops, the two channels can have completely individual structures.

Listing 5-7. Software-Triggered Descriptor Fetch

```
.import "descriptors.h";
```

```
#define N 4
.section L1_data_a;
.byte2 arrSource1[N] = { 0x1001, 0x1002, 0x1003, 0x1004 };
.byte2 arrSource2[N] = { 0x2001, 0x2002, 0x2003, 0x2004 };
.byte2 arrSource3[N] = { 0x3001, 0x3002, 0x3003, 0x3004 };
.byte2 arrDest1[N];
.byte2 arrDest2[2*N];

.struct dma_desc_list descSource1 = {
    descSource2, arrSource1,
    WDSIZE_16 | DMAEN,
    length(arrSource1), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descSource2 = {
    descSource3, arrSource2,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_16 | DMAEN,
    length(arrSource2), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descSource3 = {
    descSource1, arrSource3,
    WDSIZE_16 | DMAEN,
    length(arrSource3), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descDest1 = {
    descDest2, arrDest1,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest1), 2,
    0, 0        /* unused values */
};
.struct dma_desc_list descDest2 = {
    descDest1, arrDest2,
```

```
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest2), 2,
    0, 0        /* unused values */
};

.section L1_code;
_main:
/* write descriptor address to next descriptor pointer */
    p0.h = hi(MDMA_S0_CONFIG);
    p0.l = lo(MDMA_S0_CONFIG);
    r0.h = hi(descDest1);
    r0.l = lo(descDest1);
    [p0 + MDMA_D0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;
    r0.h = hi(descSource1);
    r0.l = lo(descSource1);
    [p0 + MDMA_S0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;

/* start first work unit */
    r6.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|DMAEN;
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    r7.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|WNR|DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;

/* wait until destination channel has finished and W1C latch */
_main.wait:
    r0 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r0, bitpos(DMA_DONE));
    if !CC jump _main.wait;
    r0.l = DMA_DONE;
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r0;

/* wait for any software or hardware event here */

/* start next work unit */
```

```
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    jump _main.wait;
_main.end:
```

Handshaked Memory DMA Example

The functional block for the handshaked MDMA operation can be considered completely separately from the MDMA channels themselves. Therefore the following HMDMA setup routine can be combined with any of the MDMA examples discussed above. Be sure that the HMDMA module is enabled before the MDMA channels.

Listing 5-8 enables the HMDMA1 block, which is controlled by the DMAR1 pin and is associated with the MDMA1 channel pair.

Listing 5-8. HMDMA1 Block Enable

```
/* optionally, enable all four bank select strobes */
    p1.l = lo(EBIU_AMGCTL);
    p1.h = hi(EBIU_AMGCTL);
    r0.l = 0x0009;
    w[p1] = r0;

/* function enable for DMAR1 */
    p1.l = lo(PORTG_FER);
    r0.l = PG12;
    w[p1] = r0;
    p1.l = lo(PORTG_MUX);
    r0.l = 0x0000;
    w[p1] = r0;

/* every single transfer requires one DMAR1 event */
    p1.l = lo(HMDMA1_BCINIT);
```

```
    r0.l = 1;
    w[p1] = r0;

/* start with balanced request counter */
    p1.l = lo(HMDMA1_ECINIT);
    r0.l = 0;
    w[p1] = r0;

/* enable for rising edges */
    p1.l = lo(HMDMA1_CONTROL);
    r2.l = REP | HMDMAEN;
    w[p1] = r2;
```

If the HMDMA is intended to copy from internal memory to external devices, the above setup is sufficient. If, however, the data flow is from outside the processor to internal memory, then this small issue must be considered—the HMDMA only controls the destination channel of the memory DMA. It does not gate requests to the source channel at all. Thus, as soon as the source channel is enabled, it starts filling the DMA FIFO immediately. In 16-bit DMA mode, this results in eight read strobes on the EBIU even before the first DMAR1 event has been detected. In other words, the transferred data and the DMAR1 strobes are eight positions off. The example in Listing 5-9 delays processing until eight DMAR1 requests have been received. By doing so, the transmitter is required to add eight trailing dummy writes after all data words have been sent. This is because the transmit channel still has to drain the DMA FIFO.

Listing 5-9. HMDMA With Delayed Processing

```
/* wait for eight requests */
    p1.l = lo(HMDMA1_ECOUNT);
    r0 = 7 (z);
initial_requests:
    r1 = w[p1] (z);
```

```
    CC = r1 < r0;
    if CC jump initial_requests;

/* disable and reenable to clear edge count */
    p1.l = lo(HMDMA1_CONTROL);
    r0.l = 0;
    w[p1] = r0;
    w[p1] = r2;
```

If the polling operation shown in Listing 5-9 is too expensive, an interrupt version of it can be implemented by using the HMDMA overflow feature. Temporarily set the HMDMAx_OVERFLOW register to eight.

Unique Information for the ADSP-BF59x Processor

Figure 5-28 on page 5-102 provides a block diagram of the ADSP-BF59x DMA controller.

The ADSP-BF59x processors do *not* contain *either* cache, an asynchronous memory interface, an SDRAM interface, *or* an HMDMA controller. Therefore, any discussion or examples above regarding cache, asynchronous memory, SDRAM, and HMDMA do *not* apply to the ADSP-BF59x.

Static Channel Prioritization

The default DMA channel priority and mapping shown in Table 5-9 on page 5-103 can be changed by altering the 4-bit PMAP field in the DMAx_PERIPHERAL_MAP registers for the peripheral DMA channels.

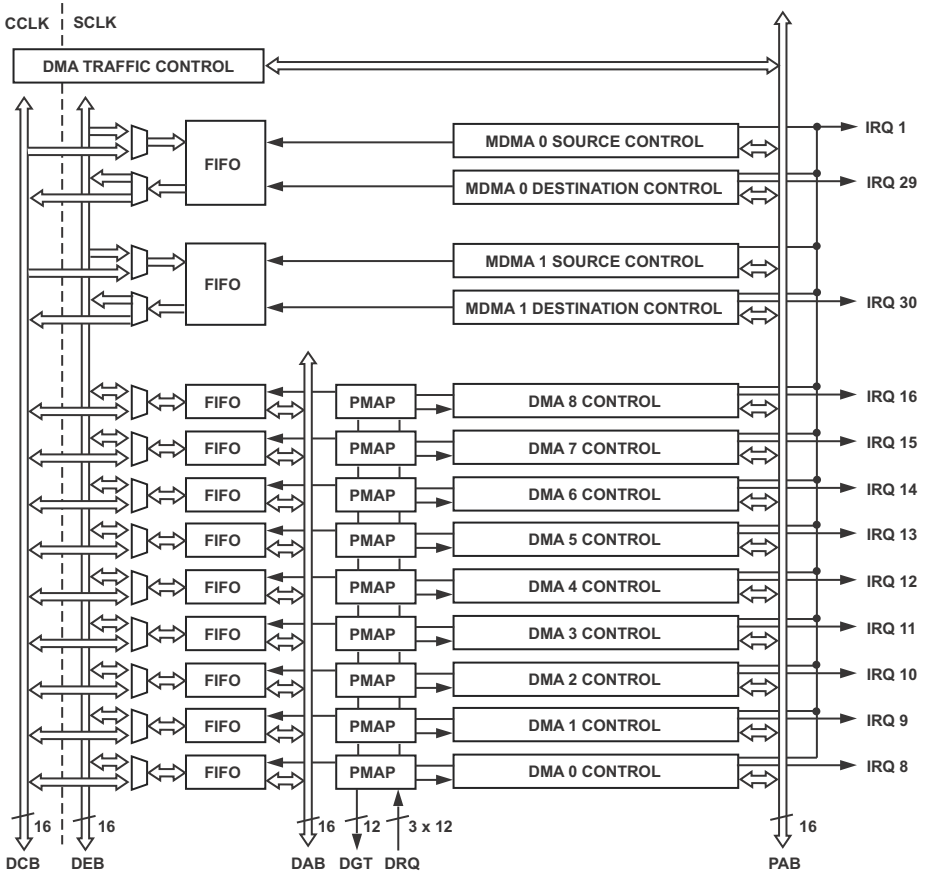


Figure 5-28. ADSP-BF59x DMA Controller Block Diagram

Table 5-9. Priority and Default Mapping of Peripheral to DMA

Priority	DMA Channel	PMAP Default Value	Peripheral Mapped by Default
Highest	DMA 0	0x0	PPI receive or transmit
	DMA 1	0x1	SPORT0 receive
	DMA 2	0x2	SPORT0 transmit
	DMA 3	0x3	SPORT1 receive
	DMA 4	0x4	SPORT1 transmit
	DMA 5	0x5	SPI0 transmit/receive
	DMA 6	0x6	SPI1 transmit/receive
	DMA 7	0x7	UART0 receive
	DMA 8	0x8	UART0 transmit
	DMA 9	0x9	Not available on this product
	DMA 10	0xA	Not available on this product
	DMA 11	0xB	Not available on this product
	MDMA D0	None	Mem DMA has no peripheral mapping.
	MDMA S0	None	Mem DMA has no peripheral mapping.
	MDMA D1	None	Mem DMA has no peripheral mapping.
Lowest	MDMA S1	None	Mem DMA has no peripheral mapping.

6 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the processor and includes the following sections:

- "Phase Locked Loop and Clock Control" on page 6-1
- "Dynamic Power Management Controller" on page 6-7
  - "Operating Modes" on page 6-7
  - "Dynamic Supply Voltage Control" on page 6-15
- "System Control ROM Function" on page 6-22
- "PLL and VR Registers" on page 6-18
- "Programming Examples" on page 6-28

Phase Locked Loop and Clock Control

The input clock into the processor, `CLKIN`, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip PLL module. During normal operation, the user programs the PLL with a multiplication factor for `CLKIN`. The resulting, multiplied signal is the voltage controlled oscillator (VCO) clock. A user-programmable value then divides the VCO clock signal to generate the core clock (`CCLK`).

A user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the Peripheral Access Bus (PAB) and DMA Access Bus (DAB).

These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL divide register, select a divider value that allows these buses to run at or below the maximum SCLK rate specified in the processor data sheet.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to be changed dynamically in a "coarse adjustment." For a "fine adjustment," the PLL clock frequency can also be varied.

PLL Overview

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the Dynamic Power Management Controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see "Dynamic Power Management Controller" on page 6-7.

Subject to the maximum VCO frequency specified in the processor data sheet, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLKIN. To achieve this wide

multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.

Figure 6-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

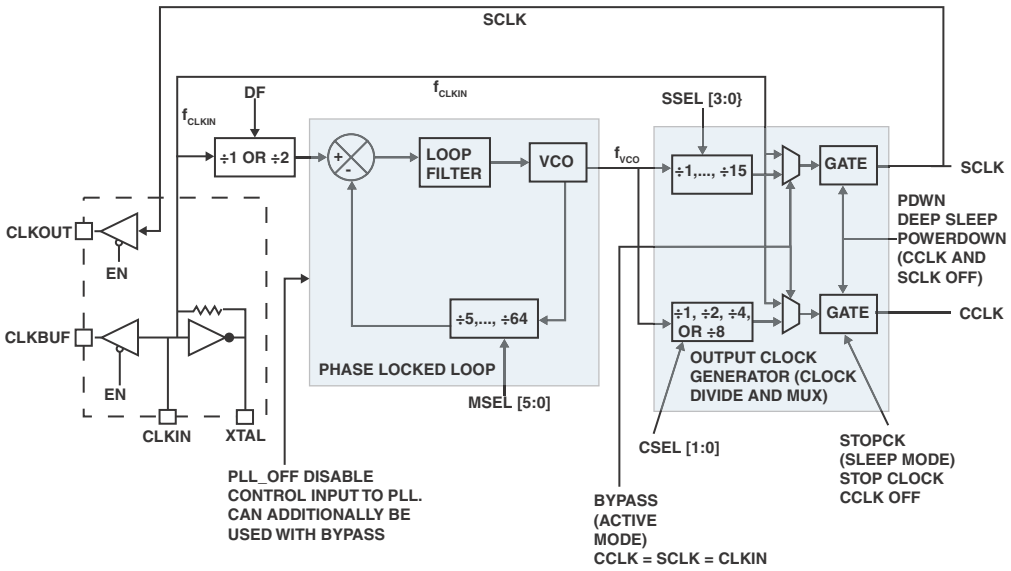


Figure 6-1. PLL Block Diagram

PLL Clock Multiplier Ratios

The PLL control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see "PLL_CTL Register" on page 6-20.

The divide frequency (DF) bit and multiplier select (MSEL[5:0]) field configure the various PLL clock dividers:

- DF enables the input divider
- MSEL[5:0] controls the feedback dividers

The reset value of MSEL is 0x6. This value can be reprogrammed at startup in the boot code.

Table 6-1 illustrates the VCO multiplication factors for the various MSEL and DF settings.

As shown in the table, different combinations of MSEL[5:0] and DF can generate the same VCO frequencies. For a given application, one combination may provide lower power or satisfy the VCO maximum frequency. Under normal conditions, setting DF to 1 typically results in lower power dissipation. See the processor data sheet for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 6-1. MSEL Encodings

Signal name	VCO Frequency	
MSEL[5:0]	DF = 0	DF = 1
5	5x	2.5x
6	6x	3x
N = 7–62	Nx	0.5Nx
63	63x	31.5x
0	64x	32x

The PLL control (PLL_CTL) register controls operation of the PLL (see Figure 6-4 on page 6-20). Note that changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with a new value, and then a specific PLL programming sequence must be executed to implement the changes. This is handled

automatically by the system control ROM function (`bfrom_SysControl()`) as described in "System Control ROM Function" on page 6-22.

Core Clock/System Clock Ratio Control

Table 6-2 describes the programmable relationship between the VCO frequency and the core clock. Table 6-3 shows the relationship of the VCO frequency to the system clock. Note the divider ratio must be chosen to limit the `SCLK` to a frequency specified in the processor data sheet. The `SCLK` drives all synchronous, system-level logic.

The divider ratio control bits, `CSEL` and `SSEL`, are in the PLL divide (`PLL_DIV`) register. For information about this register, see "PLL_DIV Register" on page 6-19.

The reset value of `CSEL[1:0]` is 0x0, and the reset value of `SSEL[3:0]` is 0x4. These values can be reprogrammed at startup by the boot code.

By updating `PLL_DIV` with an appropriate value, you can change the `CSEL` and `SSEL` value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the `PLL_DIV` register is programmed to illegal values, the `SCLK` divider is automatically increased to be greater than or equal to the core clock divider.

Unlike writing the `PLL_CTL` register, the `PLL_DIV` register can be programmed at any time to change the `CCLK` and `SCLK` divide values without entering the PLL programing sequence.

Table 6-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz) VCO	CCLK
00	1	300	300
01	2	300	150
10	4	400	100
11	8	400	50

As long as the MSEL and DF control bits in the PLL control (PLL_CTL) register remain constant, the PLL is locked.

Table 6-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz) VCO	SCLK
0000	Reserved	N/A	N/A
0001	1:1	50	50
0010	2:1	150	75
0011	3:1	150	50
0100	4:1	200	50
0101	5:1	300	60
0110	6:1	360	60
N = 7–15	N:1	400	400/N

If changing the clock ratio via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency. The PLL lock count (PLL_LOCKCNT) register defines the number of CLKIN cycles that occur before the processor sets the PLL_LOCKED bit in the PLL_STAT register. When executing the PLL programming sequence, the internal PLL lock counter begins incrementing upon execution of the IDLE instruction. The lock counter increments by 1 each CLKIN cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See the processor data sheet for more information about PLL stabilization time and programmed values for this register. For more information about operating modes, see “Operating Modes” on page 6-7.

Dynamic Power Management Controller

The Dynamic Power Management Controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor's performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See "Operating Modes" on page 6-7.
- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.
- Voltage control – The V_{DDINT} domain must be powered by an external voltage regulator. For more information see "Voltage Regulation Interface" on page 17-9.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 6-4 summarizes the operational characteristics of each mode.

Table 6-4. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled [1]	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

1 PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The active and full-on states of the DPMC/PLL can be determined by reading the PLL status register (see "PLL_STAT Register" on page 6-20). In these modes, the core can either execute instructions or be in the IDLE core state. If the core is in the IDLE state, it can be awakened by several sources (See Chapter 4, "System Interrupts" for details).

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full-On Mode

Full-on mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full-on mode is the normal execution state of the processor, with the processor and all enabled peripherals running at full speed. The system clock (SCLK) frequency is determined by the SSEL specified ratio to VCO. DMA access is available to L1 and external memories. From full-on mode, the processor can transition directly to active, sleep, or deep sleep modes, as shown in Figure 6-2 on page 6-12.

Active Mode

In active mode, the PLL is enabled but bypassed. Because the PLL is bypassed, the processor's core clock (CCLK) and system clock (SCLK) run at the input clock (CLKIN) frequency. DMA access is available to appropriately configured L1 and external memories.

In active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to full-on or sleep modes.

From active mode, the processor can transition directly to full-on, sleep, or deep sleep modes.

In this mode or in the transition phase to other modes, changes to MSEL are not latched by the PLL.

Sleep Mode

Sleep mode significantly reduces power dissipation by idling the processor core. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. Since CCLK is disabled, DMA access is available only to external memory in sleep mode. From sleep mode, a wakeup event causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full-on mode if the BYPASS bit is cleared

When sleep mode is exited, the processor resumes execution from the program counter value present immediately prior to entering sleep mode.

Deep Sleep Mode

Deep sleep mode maximizes power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals (except those enabled as wakeup sources) are disabled. DMA is not supported in this mode.

Deep sleep mode can be exited only by a hardware reset event or a wakeup event on a programmable flag pin (including PF8, PF0, PG12, or PG1). A hardware reset begins the hardware reset sequence. For more information about hardware reset, see Chapter 4, “System Interrupts”. A programmable flag event causes the processor to transition to active mode, and execution resumes at the program counter value at which the processor entered deep sleep mode. If an interrupt is also enabled in SIC_IMASK, the interrupt is vectored immediately after exit of deep sleep, and the related ISR executed.

Note that a programmable flag event in deep sleep mode automatically resets some fields in the PLL control (PLL_CTL) register. See Table 6-5.

Table 6-5. PLL_CTL Values after Programmable Flag Event

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (V_{DDINT}) to be powered down by the external regulator, while keeping the I/O supply (V_{DDEXT}) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such in the diagram of Figure 6-2.This feature is discussed in detail in "Powering Down the Core (Hibernate State)" on page 6-17.

Operating Mode Transitions

Figure 6-2 graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes and rectangles represent processor states. Arrows show the allowed transitions into and out of each mode or state.

For mode transitions, the text next to each transition arrow shows the fields in the PLL control (PLL_CTL) register that must be changed for the transition to occur. For example, the transition from full-on mode to sleep mode indicates that the STOPCK bit must be set to 1 and the PDWN bit must be set to 0.

For transitions to processor states, the text next to each transition arrow shows either a processor event (hardware reset or wakeup event) or the fields in the voltage regulator control register (VR_CTL) that must be changed for the transition to occur.

For information about how to effect mode transitions, see "Programming Operating Mode Transitions" on page 6-13.

In addition to the mode transitions shown in Figure 6-2, the PLL can be modified while in active operating mode. Changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to take effect (see "Programming Operating Mode Transitions" on page 6-13).

- PLL disabled: In addition to being bypassed in the active mode, the PLL can be disabled.

  When the PLL is disabled, additional power savings are achieved although they are relatively small. To disable the PLL, set the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- PLL enabled: When the PLL is disabled, it can be re-enabled later when additional performance is required.

  The PLL must be re-enabled before transitioning to the full-on or sleep operating modes. To re-enable the PLL, clear the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- New multiplier ratio: The multiplier ratio can also be changed while in full-on mode.

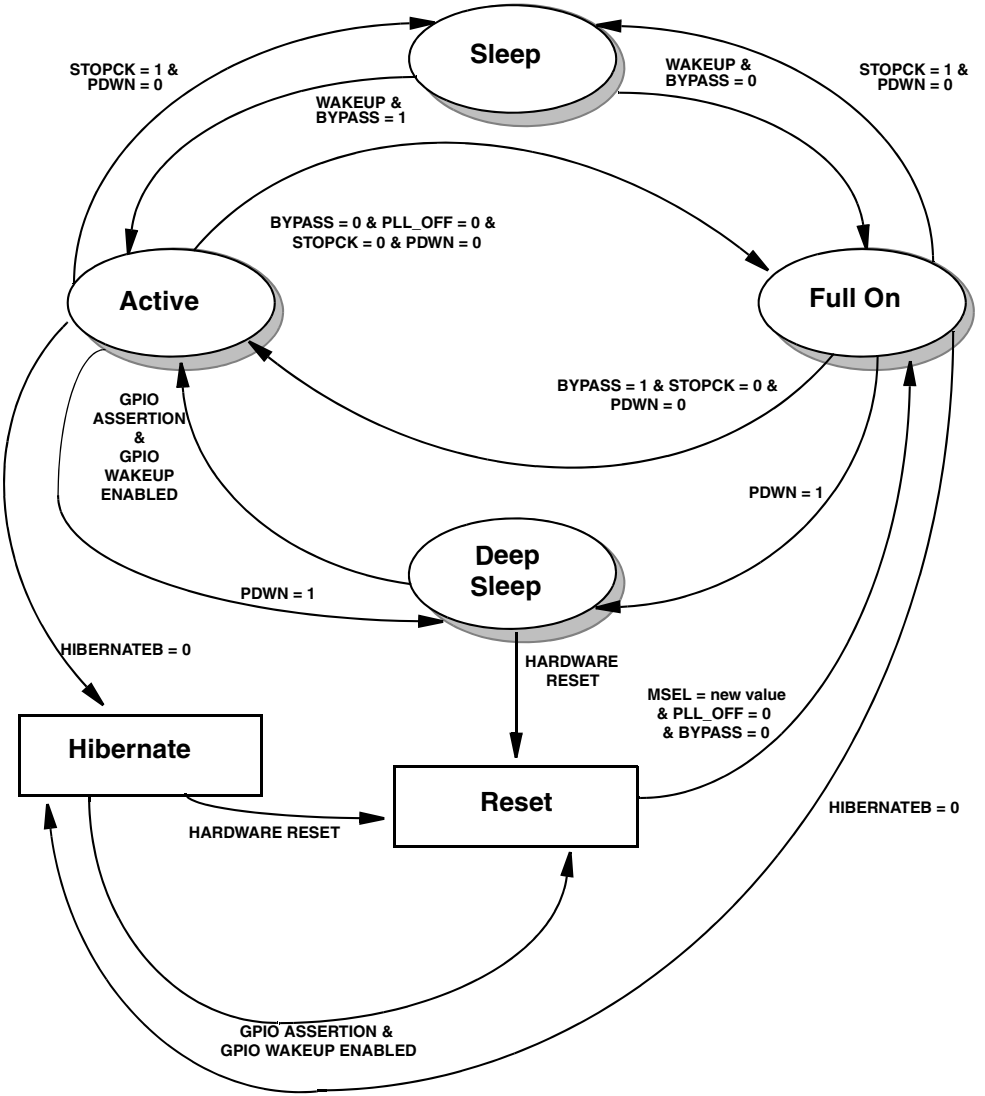


Figure 6-2. Operating Mode Transitions

The PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full-on mode. To program a new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence (see on page 6-13).

Table 6-6 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 6-6 causes unpredictable behavior.

Table 6-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full-On	Active	Sleep	Deep Sleep
Full On	–	Allowed	Allowed	Allowed
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL control (PLL_CTL) register. Merely modifying the bits of the PLL_CTL register does not change the operating mode or behavior of the PLL. Changes to the PLL_CTL register are realized only after a specific code sequence is executed. This sequence is managed by a user-callable routine in the on-chip ROM called bfrom_SysControl(). When calling this function, no further precautions have to be taken. See "System Control ROM Function" on page 6-22 for more information.

If the PLL_CTL register changes include a new CLKIN to VCO multiplier or power is reapplied to the PLL, the PLL needs to relock. To relock, the PLL lock counter is cleared first, then starts incrementing once per SCLK

cycle. After the PLL lock counter reaches the value programmed in the PLL lock count (PLL_LOCKCNT) register, the PLL sets the PLL_LOCKED bit in the PLL status (PLL_STAT) register, and the PLL asserts the PLL wake-up interrupt.

When the bfrom_SysControl() routine reprograms the PLL_CTL register with a new value, the bfrom_SysControl() routine executes a subsequent IDLE instruction and prevents all other system interrupt sources, other than the DPMC, from waking up the core from the IDLE state. If the lock counter expires, the PLL issues an interrupt, and the code execution continues the instruction after the IDLE instruction. Therefore, the system is in the new state by the time the bfrom_SysControl() routine returns.

If the new value written to the PLL_CTL or VR_CTL register is the same as the previous value, the PLL wake-up occurs immediately (PLL is already locked), but the core and system clock are bypassed for the PLL_LOCKCNT duration. For this interval, code executes at the CLKIN rate instead of the expected CCLK rate. Software guards against this condition by comparing the current value to the new value before writing the new value.

- When the wake-up signal is asserted, the code execution continues the instruction after the IDLE instruction, causing a transition to:
    - Active mode if BYPASS in the PLL_CTL register is set
    - Full-on mode if the BYPASS bit is cleared
- If the PLL_CTL register is programmed to enter the sleep operating mode, the processor transitions immediately to sleep mode and waits for a wake-up signal before continuing code execution. If the PLL_CTL register is programmed to enter the deep sleep operating mode, the processor immediately transitions to deep sleep mode and waits for a hardware reset signal or GPIO wakeup:

- A hardware reset causes the processor to execute the reset sequence. For more information, see “System Reset and Booting” on page 16-1.
- A GPIO wakeup event causes the processor to enter active operating mode and return from the `bfrom_SysControl()` routine.

If no operating mode transition is programmed, the PLL generates a wake-up signal, and the `bfrom_SysControl()` routine returns.

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor's core is capable of running at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses multiple power domains. Each power domain has a separate V_{DD} supply. Note that the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See the product data sheet for details on the allowed voltage ranges for each power domain and power dissipation data.

Power Supply Management

V_{DDINT} is supplied by an external regulator and pin $\overline{PG}$ is used to accept an active-low power-good indicator from the regulator. Note that the external regulator must comply with the V_{DDINT} specifications defined in the processor data sheet.

Changing Voltage

When changing the voltage using an external regulator, a specific programming sequence must be followed.

Unlike other Blackfin derivatives that feature an internal voltage regulator; the voltage level for the ADSP-BF59x cannot be changed by programming the VR_CTL register. With an internal voltage regulator, the PLL would automatically enter the active mode when the processor enters the IDLE state. At that point the voltage level would change and the PLL would re-lock to the new voltage. After the PLL_LOCKCNT has expired, the part returns to the full-on state.

With an external voltage regulator, this sequence must be reproduced in the program code by the user. The PLL_LOCKCNT register cannot be used in this case, but the value is still needed for calculating the required delay. A larger PLL_LOCKCNT value may be necessary for changing voltages than when changing just the PLL frequency. See the processor data sheet for details.

The processor must enter active mode before the user can access the external voltage regulator and program a new voltage level. See the data sheet of external voltage regulator for information on changing voltage levels. See the processor data sheet for more information about voltage tolerances and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior the recommended procedure is to bring the processor to the sleep operating mode before substantially varying the voltage.

The user must ensure a stable voltage and give the PLL time to re-lock at the new voltage level. This can be done by running the core in a loop for a certain amount of time before leaving active mode.

After the voltage has been changed to the new level, the processor can safely return to any operational mode—so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in the processor data sheet for the new operating voltage level.

Please see "Changing Voltage Levels" on page 6-38 for more details on mode transitions and changing voltage levels.

The VSTAT bit in the PLL_STAT register can be used to indicate whether V_{DDINT} is stable and ready to use. The VSTAT bit works in conjunction with the $\overline{PG}$ (Power Good) input signal of the ADSP-BF59x. The inverted version of a "power good" signal from the external regulator is fed to the ADSP-BF59x to indicate that the voltage has reached its programmed value. That in turn will set the VSAT bit, which should be considered the end of your "wait" state for the voltage regulator to settle.

Powering Down the Core (Hibernate State)

The external regulator can be signaled to shut off V_{DDINT} using the EXT_WAKE signal. Writing 0 to the HIBERNATEB bit of the VR_CTL register, which disables CCLK and SCLK, will also make EXT_WAKE go low. EXT_WAKE will transition high if any wakeup sources occur, which will signal the external voltage regulator to turn V_{DDINT} on again. The wakeup sources are several user-selectable events, all of which are controlled in the VR_CTL register:

- Assertion of the $\overline{RESET}$ pin always exits hibernate state and requires no modification to VR_CTL.
- External GPIO event. Set a GPIO wakeup enable control bit (WAKE_EN0, WAKE_EN1, WAKE_EN2, WAKE_EN3) to enable wakeup on assertion of a signal on the corresponding pin.
- Pin EXT_WAKE is provided to indicate the occurrence of wakeup. EXT_WAKE is an output pin, which is a logical OR of the above wakeup sources, except hardware reset. The pin follows the wakeup signal of the various wakeup sources.

When the core is powered down, V_{DDINT} is set to 0 V, and the internal state of the processor is not maintained, with the exception of the VR_CTL register. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power.

Powering down V_{DDINT} does not affect V_{DDEXT}. While V_{DDEXT} is still applied to the processor, external pins are maintained at a three-state level unless specified otherwise.

To signal the external regulator to power down V_{DDINT}:

1. Write 0 to the appropriate bits in the SIC_IWRx registers to prevent enabled peripheral resources from interrupting the hibernate process.
2. Call the bfrom_SysControl() routine; ensure that the HIBERNATEB bit in the VR_CTL register is set to 0, and the appropriate wakeup enable bit or bits (WAKE_EN0, WAKE_EN1, WAKE_EN2, or WAKE_EN3) are set to 1.
3. The bfrom_SysControl() routine executes until V_{DDINT} transitions to 0 V. The bfrom_SysControl() routine never returns.
4. When the processor is woken up, the PLL relocks and the boot sequence defined by the BMODE[2:0] pin settings takes effect.

The WURESET bit in the SYSCTRL register is set and stays set until the next hardware reset. The WURESET bit may control a conditional boot process.

PLL and VR Registers

The user interface to the PLL and VR registers is through the system control ROM function (bfrom_SysControl()) described in "System Control ROM Function" on page 6-22. The memory-mapped registers (MMRs) are shown in Table 6-7 and illustrated in Figure 6-3 through Figure 6-7.

Table 6-7 shows the functions of the PLL/VR registers.

Table 6-7. PLL/VR Register Mapping

Register Name	Function	Notes	For More Information See:
PLL_CTL	PLL control register	Requires reprogramming sequence when written	Figure 6-4 on page 6-20
PLL_DIV	PLL divisor register	Can be written freely	Figure 6-3 on page 6-19
PLL_STAT	PLL status register	Monitors active modes of operation	Figure 6-5 on page 6-20
PLL_LOCKCNT	PLL lock count register	Number of SCLKs allowed for PLL to relock	Figure 6-6 on page 6-21
VR_CTL	Voltage regulator control register	Requires PLL reprogramming sequence when written	Figure 6-7 on page 6-21

PLL_DIV Register

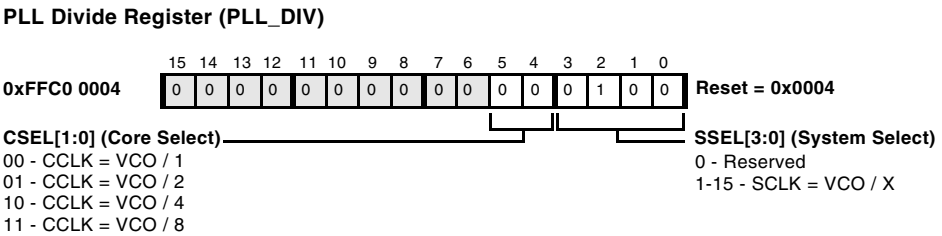
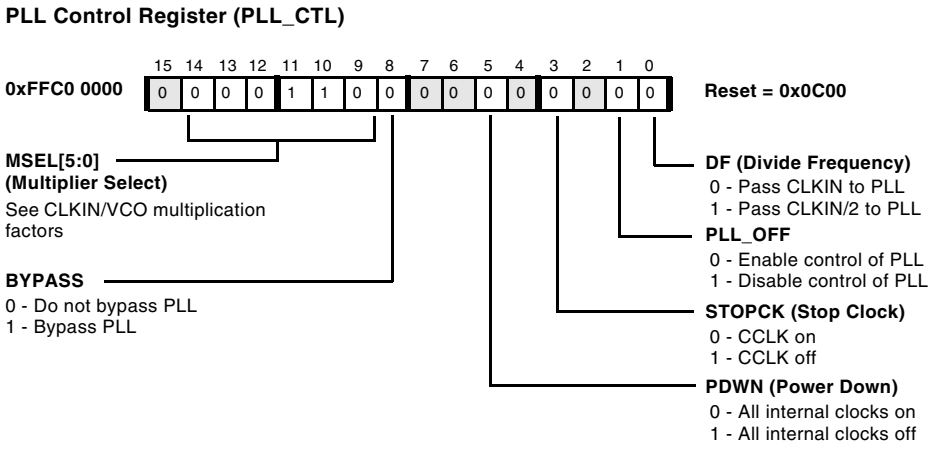

PLL Divide Register (PLL_DIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0004
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
Reset = 0x0004
CSEL[1:0] (Core Select)
00 - CCLK = VCO / 1
01 - CCLK = VCO / 2
10 - CCLK = VCO / 4
11 - CCLK = VCO / 8
SSEL[3:0] (System Select)
0 - Reserved
1-15 - SCLK = VCO / X

Figure 6-3. PLL Divide Register

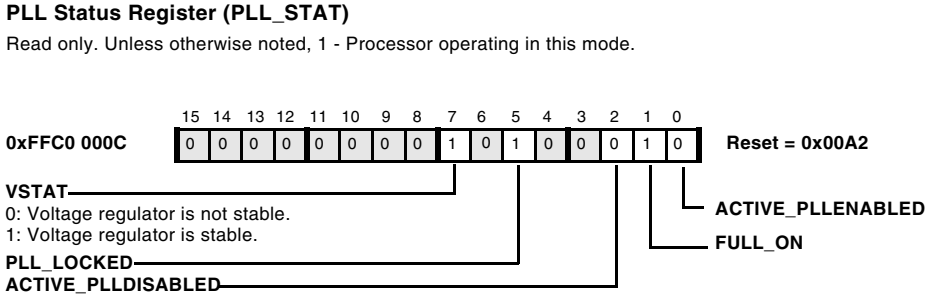
PLL_CTL Register


PLL Control Register (PLL_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0000
0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
Reset = 0x0C00
MSEL[5:0]
(Multiplier Select)
See CLKIN/VCO multiplication factors
BYPASS
0 - Do not bypass PLL
1 - Bypass PLL
DF (Divide Frequency)
0 - Pass CLKIN to PLL
1 - Pass CLKIN/2 to PLL
PLL_OFF
0 - Enable control of PLL
1 - Disable control of PLL
STOPCK (Stop Clock)
0 - CCLK on
1 - CCLK off
PDWN (Power Down)
0 - All internal clocks on
1 - All internal clocks off

For CLKIN/VCO multiplication factors, see Table 6-1 on page 6-4.

Figure 6-4. PLL Control Register

PLL_STAT Register


PLL Status Register (PLL_STAT)
Read only. Unless otherwise noted, 1 - Processor operating in this mode.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 000C
0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0
Reset = 0x00A2
VSTAT
0: Voltage regulator is not stable.
1: Voltage regulator is stable.
PLL_LOCKED
ACTIVE_PLLDISABLED
ACTIVE_PLLENABLED
FULL_ON

For more information, see “Operating Modes” on page 6-7.

Figure 6-5. PLL Status Register

PLL_LOCKCNT Register

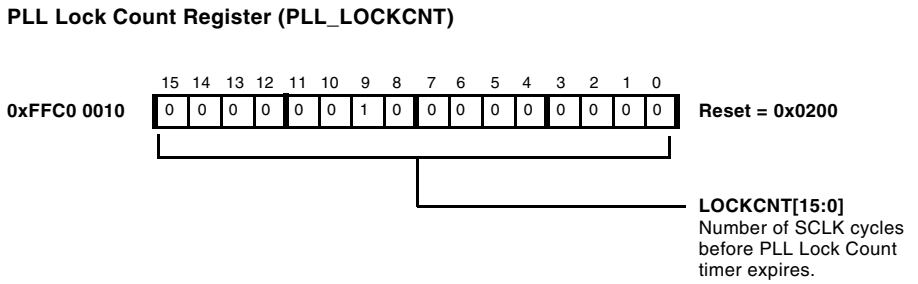

PLL Lock Count Register (PLL_LOCKCNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0010
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
Reset = 0x0200
LOCKCNT[15:0]
Number of SCLK cycles
before PLL Lock Count
timer expires.

Figure 6-6. PLL Lock Count Register

VR_CTL Register

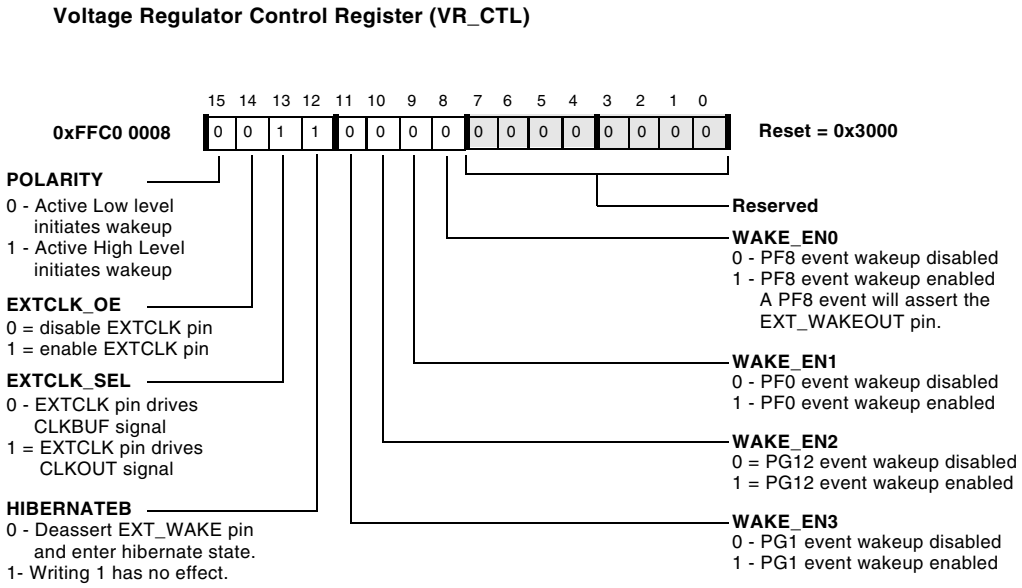

Voltage Regulator Control Register (VR_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0008
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x3000
POLARITY
0 - Active Low level
initiates wakeup
1 - Active High Level
initiates wakeup
EXTCLK_OE
0 = disable EXTCLK pin
1 = enable EXTCLK pin
EXTCLK_SEL
0 - EXTCLK pin drives
CLKBUF signal
1 = EXTCLK pin drives
CLKOUT signal
HIBERNATEB
0 - Deassert EXT_WAKE pin
and enter hibernate state.
1- Writing 1 has no effect.
Reserved
WAKE_EN0
0 - PF8 event wakeup disabled
1 - PF8 event wakeup enabled
A PF8 event will assert the
EXT_WAKEOUT pin.
WAKE_EN1
0 - PF0 event wakeup disabled
1 - PF0 event wakeup enabled
WAKE_EN2
0 = PG12 event wakeup disabled
1 = PG12 event wakeup enabled
WAKE_EN3
0 - PG1 event wakeup disabled
1 - PG1 event wakeup enabled

Figure 6-7. Voltage Regulator Control Register

The external clock select (EXTCLK_SEL) control bit configures the EXTCLK

pin to output either the SCLK frequency (called CLKOUT) or to output an input buffered CLKIN frequency (called CLKBUF). When configured to output SCLK (CLKOUT), the EXTCLK pin acts as a reference signal in many timing specifications. When configured to output CLKIN (CLKBUF), the EXTCLK pin allows another device *and* the processor to run from a single crystal oscillator.

The external clock output enable (EXTCLK_OE) control bit configures the EXTCLK pin to *either* enable (when set, =1) *or* disable (when cleared, =0) the output of the clock signal selected by EXTCLK_SEL. When EXTCLK_OE is cleared, the EXTCLK pin is three-stated.

The POLARITY control bit configure the active level of the wakeup event on the programmable flags.

System Control ROM Function

The PLL and voltage regulator registers should not be accessed directly. Instead, use the bfrom_SysControl() function to alter or read the register values. The function resides in the on-chip ROM and can be called by the user following C-language style calling conventions.

Entry address: 0xEF00 0038

Arguments:

- dActionFlags word in R0
- pSysCtrlSettings pointer in R1
- zero value in R2

A potential error message of internally called bfrom_OtpRead() function forwarded and returned in R0.

The system control ROM function does not verify the correctness of the forwarded arguments. Therefore, it is up to the programmer to choose the correct values.

C prototype: `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);`

The first argument (`u32 dActionFlags`) to the system control ROM function holds the instruction flags. The following flags are supported.

```
#define SYSCTRL_READ          0x00000000
#define SYSCTRL_WRITE         0x00000001
#define SYSCTRL_SYSRESET      0x00000002
#define SYSCTRL_SOFTRESET     0x00000004
#define SYSCTRL_VRCTL         0x00000010
#define SYSCTRL_EXTVOLTAGE    0x00000020
#define SYSCTRL_PLLCTL        0x00000100
#define SYSCTRL_PLLDIV        0x00000200
#define SYSCTRL_LOCKCNT       0x00000400
#define SYSCTRL_PLLSTAT       0x00000800
```

With `SYSCTRL_READ` and `SYSCTRL_WRITE`, a read or a write operation is initialized. The `SYSCTRL_SYSRESET` flag performs a system reset, while the `SYSCTRL_SOFTRESET` flag combines a core and system reset. The `SYSCTRL_EXTVOLTAGE` flag indicates that V_{DDINT} is supplied externally. The last five flags (`_VRCTL`, `_PLLCTL`, `_PLLDIV`, `_LOCKCNT`, `_PLLSTAT`) tells the system control ROM function which registers to be written to or read from. Note that `SYSCTRL_PLLSTAT` flag is read-only.

The second argument (`ADI_SYSCTRL_VALUES *pSysCtrlSettings`) to the system control ROM function passes a pointer to a special structure, which has entries for all PLL and voltage regulator registers. It is predefined in the `bfrom.h` header file as follows.

```
typedef struct
{
   u16 uwVrCtl;
   u16 uwPllCtl;
   u16 uwPllDiv;
   u16 uwPllLockCnt;
   u16 uwPllStat;
} ADI_SYSCTRL_VALUES;
```

The third argument to the system control ROM function is reserved and should be kept zero (NULL pointer).

The function's return value is described in the following `bfrom_OtpRead()` ROM routine descriptions; whereby a single-bit warning is suppressed.

The system control ROM function executes the correct steps and programming sequence for the Dynamic Power Management System of the Blackfin processor.

Programming Model

The programming model for the system control ROM function in C/C++ and Assembly is described in the following sections.

Accessing the System Control ROM Function in C/C++

To read the `PLL_DIV` and `PLL_CTL` register values, for example, specify the `SYSCTRL_READ` instruction flag along with the `SYSCTRL_PLLCTL` and `SYSCTRL_PLLDIV` register flags. The `bfrom_OtpRead()` function then only updates the `uwPllCtl` and `uwPllDiv` variables:

```
ADI_SYSCTRL_VALUES read;
bfrom_SysControl (SYSCTRL_READ | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV,
&read, NULL);
```

The read.uwPllCtl and read.uwPllDiv variables access the PLL_CTL and PLL_DIV register values, respectively. To update the register values, specify the SYSCTRL_WRITE instruction flag along with the register flags of those registers that should be modified and have valid data in the respective ADI_SYSCTRL_VALUES variables:

```
ADI_SYSCTRL_VALUES write;
write.uwPllCtl = 0x0C00;
write.uwPllDiv = 0x0004;
bfrom_SysControl (SYSCTRL_WRITE | SYSCTRL_PLLCTL |SYSCTRL_PLLDIV,
&write, NULL);
```

Accessing the System Control ROM Function in Assembly

The assembler supports C structs, which is required to import the file bfrom.h:

```
#include <bfrom.h>
.IMPORT "bfrom.h";
.STRUCT ADI_SYSCTRL_VALUES dpm;
```

You can pre-load the struct:

```
.STRUCT ADI_SYSCTRL_VALUES dpm = { 0x3000, 0x0C00, 0x0004,
0x0200, 0x00A2 };
```

or load the values dynamically inside the code:

```
P5.H = hi(dpm);

P5.L = lo(dpm->uwVrCtl);
R7 = 0x3000 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllCtl);
```

```
R7 = 0x0C00 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllDiv);
R7 = 0x0004 (z);
w[P5] = R7;

P5.L = lo(dpm->uwPllLockCnt);
R7 = 0x0200 (z);
w[P5] = R0;
```

The function `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);` can be accessed by `BFROM_SYSCONTROL`. Following the C/C++ run-time environment conventions, the parameters passed are hold by the data registers `R0`, `R1`, and `R2`.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */

link sizeof(ADI_SYSCTRL_VALUES)+2;

[--SP] = (R7:0,P5:0);

/* Allocate at least 12 bytes on the stack for outgoing argu-
ments, even if the function being called requires less than this.
*/
SP += -12;

R0 = SYSCTRL_WRITE      |
     SYSCTRL_VRCTL      |
     SYSCTRL_EXTVOLTAGE |
     SYSCTRL_PLLCTL     |
     SYSCTRL_PLLDIV     ;
```

```
R1.H = hi(dpm);
R1.L = lo(dpm);
R2 = 0 (z);
P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

The processor's internal scratchpad memory can be used as an alternative for taking a C struct. Therefore, the stack/frame pointer must be loaded and passed.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;

[--SP] = (R7:0,P5:0);

/* Allocate at least 12 bytes on the stack for outgoing argu-
ments, even if the function being called requires less than this.
*/
SP += -12;

R7 = 0;
R7.L = 0x3000;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R7;
R7.L = 0x0C00;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R7;
```

```
R7.L = 0x0004;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R7;
R7.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R7;

R0 = SYSCTRL_WRITE      |
     SYSCTRL_VRCTL      |
     SYSCTRL_EXTVOLTAGE |
     SYSCTRL_PLLCTL     |
     SYSCTRL_PLLDIV     ;
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0;

P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

Programming Examples

The following code examples illustrate how to use the system control ROM function to effect various operating mode transitions.

The following examples are only meant to demonstrate how to program the PLL registers. Do not assume that the voltages and frequencies shown in the examples are supported by your processor. Instead, check your product's data sheet for supported voltages and frequencies.

Some setup code has been removed for clarity, and the following assumptions are made.

- PLL control (`PLL_CTL`) register setting: 0x0C00
- PLL divider (`PLL_DIV`) register setting: 0x0004
- PLL lock count (`PLL_LOCKCNT`) register setting: 0x0200
- Clock in (`CLKIN`) frequency: 25 MHz

VCO frequency is 125 MHz, core clock frequency is 125 MHz, and system clock frequency is 31.25 MHz.

- Voltage regulator control (`VR_CTL`) register setting: 0x3000
- Logical voltage level (`VDDINT`) is at 1.20 V

For operating mode transition and voltage regulator examples:

- **C**
    - `#include  <blackfin.h>`
    - `#include <bfrom.h>`
- **Assembly**
    - `#include  <blackfin.h>`
    - `#include <bfrom.h>`
    - `.IMPORT "bfrom.h";`

- `#define IMM32(reg,val) reg##.H=hi(val);`
- `reg##.L=lo(val);`

Full-on Mode to Active Mode and Back

Listing 6-1 and Listing 6-2 provide code for transitioning from the full-on operating mode to active mode in C and Blackfin assembly code, respectively.

Listing 6-1. Transitioning from Full-on Mode to Active Mode (C)

```
void active(void)
{
ADI_SYSCTRL_VALUES active;
bfrom_SysControl(SYSCTRL_READ | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL, &active, NULL);
active.uwPllCtl |= (BYPASS | PLL_OFF); /* PLL_OFF bit optional */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL, &active, NULL);
return;
}
```

Listing 6-2. Transitioning from Full-on Mode to Active Mode (ASM)

```
__active:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = (SYSCTRL_READ | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
```

```
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

R0 =
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(BYPASS));
bitset(R0,bitpos(PLL_OFF)); /* optional */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0 = (SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__active.end:
```

To return from active mode (go back to full-on mode), the `BYPASS` bit and the `PLL_OFF` bit must be cleared again, respectively.

Transition to Sleep Mode or Deep Sleep Mode

Listing 6-3 and Listing 6-4 provide code for transitioning from the full-on operating mode to sleep or deep sleep mode in C and Blackfin assembly code, respectively.

Listing 6-3. Transitioning to Sleep Mode or Deep Sleep Mode (C)

```
void sleep(void)
{
ADI_SYSCTRL_VALUES sleep;
bfrom_SysControl(SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &sleep, NULL);
sleep.uwPllCtl |= STOPCK;    /* either: Sleep Mode */
sleep.uwPllCtl |= PDWN;      /* or: Deep Sleep Mode */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL, &sleep, NULL);
return;
}
```

Listing 6-4. Transitioning to Sleep Mode or Deep Sleep Mode (ASM)

```
__sleep:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = (SYSCTRL_READ | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
```

```
call(P4);

R0 =
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)];
bitset(R0,bitpos(STOPCK)); /* either: Sleep Mode */
bitset(R0,bitpos(PDWN)); /* or: Deep Sleep Mode */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0 = (SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__sleep.end:
```

Set Wakeup Events and Enter Hibernate State

Listing 6-5 and Listing 6-6 provide code for configuring the regulator wakeups (PF8, PF0, PG12, and PG1) and placing the regulator in the hibernate state in C and processor assembly code, respectively.

Listing 6-5. Configuring Regulator Wakeups and Entering Hibernate

State (C)

```
void hibernate(void)
{
ADI_SYSCTRL_VALUES hibernate;
hibernate.uwVrCtl=WAKE_EN0 | /* PF8 Wake-Up Enable */
WAKE_EN1 | /* PF0 Wake-Up Enable */
WAKE_EN2 | /* PG12 Wake-Up Enable */
WAKE_EN3 | /* PG1 Wake-Up Enable */
HIBERNATE; / *Powerdown */
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_VRCTL |
SYSCTRL_EXTVOLTAGE, &hibernate, NULL);
/* Hibernate State: no code executes until wakeup triggers
reset */
}
```

Listing 6-6. Configuring Regulator Wakeups and Entering Hibernate State (ASM)

```
__hibernate:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;
cli R6; /* disable interrupts, copy IMASK to R6 */
R0.L = WAKE_EN0 | /* PF8 Wake-Up Enable */
WAKE_EN1 | /* PF0 Wake-Up Enable */
WAKE_EN2 | /* PG12 Wake-Up Enable */
WAKE_EN3 | /* PG1 Wake-Up Enable */
HIBERNATE; / *Powerdown */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwVrCtl)] = R0;
R0 = (SYSCTRL_WRITE | SYSCTRL_VRCTL | SYSCTRL_EXTVOLTAGE);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
```

```
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);
/* Hibernate State: no code executes until wakeup triggers
reset */
__hibernate.end:
```

Note that there may be a need to call bfrom_SysControl() twice, once to setup the polarity and the wakeup sources and once to enter hibernate.

Perform a System Reset or Soft-Reset

Listing 6-7 and Listing 6-8 provide code for executing a system reset *or* a soft-reset (system and core reset) in C and Blackfin assembly code, respectively.

Listing 6-7. Execute a System Reset or a Soft-Reset (C)

```
void reset(void)
{
bfrom_SysControl(SYSCTRL_SYSRESET, NULL, NULL); /* either */
bfrom_SysControl(SYSCTRL_SOFTRESET, NULL, NULL); /* or */
return;
}
```

Listing 6-8. Execute a System Reset or a Soft-Reset (ASM)

```
__reset:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = SYSCTRL_SYSRESET; /* either */
```

```
R0 = SYSCTRL_SOFTRESET; /* or */
R1 = 0 (z);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__reset.end:
```

In Full-on Mode, Change VCO Frequency, Core Clock Frequency, and System Clock Frequency

Listing 6-9 and Listing 6-10 provide C and Blackfin assembly code for changing the CLKIN to VCO multiplier (from 10x to 21x), keeping the CSEL divider at 1, and changing the SSEL divider (from 5 to 4) in the full-on operating mode.

Listing 6-9. Transition of Frequencies (C)

```
void frequency(void)
{
ADI_SYSCTRL_VALUES frequency;

/* Set MSEL = 5-63 --> VCO = CLKIN*MSEL */
frequency.uwPllCtl = SET_MSEL(21) ;

/* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
/* CCLK = VCO / 1 */
frequency.uwPllDiv = SET_SSEL(4) |
                     CSEL_DIV1   ;
```

```
frequency.uwPllLockCnt = 0x0200;

bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE |
SYSCTRL_PLLCTL | SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT |, &frequency,
NULL);
return;
}
```

Listing 6-10. Transition of Frequencies (ASM)

```
__frequency:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

/* write the struct */
R0 = 0;

R0.L = SET_MSEL(21) ; /* Set MSEL = 5-63 --> VCO = CLKIN*MSEL */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;

R0.L = SET_SSEL(4) | /* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
       CSEL_DIV1   ; /* CCLK = VCO / 1 */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;

R0.L = 0x0200;
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;

/* argument 1 in R0 */
```

```
R0 = (SYSCTRL_WRITE | SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_PLLDIV);

/* argument 2 in R1: structure lays on local stack */
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);

/* argument 3 must always be NULL */
R2 = 0;

/* call of SysControl function */
IMM32(P4,BFROM_SYSCONTROL);
call (P4); /* R0 contains the result from SysControl */

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__frequency.end:
```

Changing Voltage Levels

Listing 6-11 provides C code for changing the voltage level dynamically. The User must include his own code for accessing the external voltage regulator.

Listing 6-11. Changing Core Voltage (C)

```
void voltage(void)
{
ADI_SYSCTRL_VALUES voltage;
u32 ulCnt = 0;
```

```
bfrom_SysControl( SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_READ, &init, NULL );
init.uwPllCtl |= BYPASS;
init.uwPllLockCnt = 0x0200;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_PLLCTL | SYSCTRL_LOCKCNT
| SYSCTRL_EXTVOLTAGE, &voltage, NULL);
/* Put your code for accessing the external voltage regulator
here */
/* A delay loop is required to ensure VDDint is stable and the
PLL has re-locked. As this is depending on the external voltage
regulator circuitry the user must ensure timings are kept. The
compiler (no optimization enabled) will create a loop that takes
about 10 cycles. Time base is CLKIN as the PLL is bypassed. We
need 0x0200 CLKIN cycles that represent PLL_LOCKCNT and addition-
ally the time required by the circuitry */
ulCnt = 0x0200 + 0x0200;
while (ulCnt != 0) {ulCnt--;}
init.uwPllCtl &= ~BYPASS;
bfrom_SysControl(SYSCTRL_WRITE | SYSCTRL_PLLCTL |
SYSCTRL_EXTVOLTAGE, &voltage, NULL);
return;
}
```

7 GENERAL-PURPOSE PORTS

This chapter describes the general-purpose ports. Following an overview and a list of key features is a block diagram of the interface and a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The ADSP-BF59x Blackfin processors feature a rich set of peripherals, which, through a powerful pin multiplexing scheme, provides great flexibility to the external application space.

Features

The peripheral pins are functionally organized into general-purpose ports designated port F and port G.

Port F provides 16 pins:

- SPORT1 signals
- PPI data and frame sync signals
- Primary SPI0 signals
- GP Timer signals
- SPI0 and SPI1 slave selects

- UART0 signals
- GPIOs

Port G provides 16 pins:

- SPORT0 signals
- Primary SPI1 signals
- SPI0 and SPI1 slave selects
- PPI data signals
- GPIOs

Note that the PPI clock and the TWI signals are provided on separate pins, independent of the ports.

Interface Overview

By default, all port F and port G pins are in general-purpose I/O (GPIO) mode. In this mode, a pin can function as a digital input, digital output, or interrupt input. See "General-Purpose I/O Modules" on page 7-8 for details. Peripheral functionality must be explicitly enabled by the function enable registers (`PORTF_FER` and `PORTG_FER`). The competing peripherals on port F and port G are controlled by the respective multiplexer control register (`PORTF_MUX` and `PORTG_MUX`).

In this chapter, the naming convention for registers and bits uses a lowercase `x` to represent F or G. For example, the name `PORTx_FER` represents `PORTF_FER` and `PORTG_FER`. The bit name `Px0` represents `PF0` and `PG0`. This convention is used to discuss registers common to these ports.

External Interface

The external interface of the general-purpose ports are described in the following sections.

Port F Structure

Table 7-1 on page 7-3 shows the multiplexer scheme for port F. Port F is controlled by the PORTF_MUX and the PORTF_FER registers.

Port F consists of 16 pins, referred to as PF0 to PF15, as shown in Table 7-1 on page 7-3. All the input signals in the "Additional Use" column are enabled by their module only, regardless of the state of the PORTx_MUX and PORTx_FER registers.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in PORTF_FER is cleared.

Table 7-1. Port F Multiplexing Scheme

PORTF_MUX	0	1		
	1st function	2nd function	Additional Use	GPIO
Bit 0	DR1SEC	PPI8	WAKEN1	PF0
Bit 1	DR1PRI	PPI9		PF1
Bit 2	RSCLK1	PPI10		PF2
Bit 3	RFS1	PPI11		PF3
Bit 4	DT1SEC	PPI12		PF4
Bit 5	DT1PRI	PPI13		PF5
Bit 6	TSCLK1	PPI14		PF6
Bit 7	TFS1	PPI15		PF7
Bit 8	TMR2	$\overline{\text{SPIOSSEL2}}$	WAKEN0	PF8
Bit 9	TMR0/PPI_FS1	$\overline{\text{SPIOSSEL3}}$		PF9
Bit 10	TMR1/PPI_FS2	Reserved		PF10

Table 7-1. Port F Multiplexing Scheme (Continued)

PORTF_MUX	0	1		
	1st function	2nd function	Additional Use	GPIO
Bit 11	UART0TX	SPI0SSEL4		PF11
Bit 12	UART0RX	SPI0SSEL7	TACI2-0	PF12
Bit 13	SPI0MOSI	SPI1SSEL3		PF13
Bit 14	SPI0MISO	SPI1SSEL4		PF14
Bit 15	SPI0CLK	SPI1SSEL5		PF15

Port G Structure

Table 7-2 on page 7-4 shows the multiplexer scheme for port G. Port G is controlled by the PORTG_MUX and PORTG_FER registers.

Port G consists of 16 pins, referred to as PG0 to PG15, as shown in Table 7-2.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTG_FER register is cleared.

Table 7-2. Port G Multiplexing Scheme

PORTG_MUX	0	1		
	1st function	2nd function	Additional Use	GPIO
Bit 0	DR0SEC	SPI0SSEL1	SPI0SS	PG0
Bit 1	DR0PRI	SPI1SSEL1	WAKEN3	PG1
Bit 2	RSCLK0	SPI0SSEL5		PG2
Bit 3	RFS0	PPI_FS3		PG3
Bit 4	DT0SEC	SPI0SSEL6		PG4/HWAIT
Bit 5	DT0PRI	SPI1SSEL6		PG5
Bit 6	TSCLK0	Reserved		PG6
Bit 7	TFS0	SPI1SSEL7		PG7

Table 7-2. Port G Multiplexing Scheme (Continued)

PORTG_MUX	0	1		
	1st function	2nd function	Additional Use	GPIO
Bit 8	SPI1CLK	PPI0		PG8
Bit 9	SPI1MOSI	PPI1		PG9
Bit 10	SPI1MISO	PPI2		PG10
Bit 11	$\overline{\text{SPI1SSEL5}}$	PPI3		PG11
Bit 12	$\overline{\text{SPI1SSEL2}}$	PPI4	WAKEN2	PG12
Bit 13	$\overline{\text{SPI1SSEL1}}$	PPI5	$\overline{\text{SPI1SS}}$	PG13
Bit 14	$\overline{\text{SPI1SSEL4}}$	PPI6	TACLK1	PG14
Bit 15	$\overline{\text{SPI1SSEL6}}$	PPI7	TACLK2	PG15

Additional Considerations

Port control and GPIO registers are part of the system memory-mapped registers (MMRs). The addresses of the GPIO module MMRs appear in Appendix A, "System MMR Assignments". Core access to the GPIO configuration registers is through the system bus. The PORTx_MUX registers control the muxing schemes of port F and port G. The function enable registers (PORTF_FER and PORTG_FER) enable the peripheral functionality for each individual pin of a port.

- If pin PF9 serves as TMR0/PPI_FS1, TMR0 is internally looped back to PPI_FS1 whenever TMR0 is configured as an output. Similarly, if pin PF10 serves as TMR1/PPI_FS2, TMR1 is internally looped back to PPI_FS2 whenever TMR1 is configured as an output. This is done to avoid possible erroneous behavior associated with ringing if outputs are not well terminated. Whenever TMR0 or TMR1 inputs are used, PPI_CLK must be separately specified as the clock input for the associated timer.

- Pins configured to serve as triggers for hibernate and deepsleep wakeup (see register VR_CTL) automatically have their input buffers enabled. Note that if there are multiple simultaneous uses of these input buffers (such as wake source and GPIO input), each use must still be individually enable to achieve reliable behavior.
- The input buffer of pin PG14 is automatically enabled if system timer 1 specifies TACLK as its input clock source.
- The input buffer of pin PG15 is automatically enabled if system timer 2 specifies TACLK as its input clock source.
- The input buffer of pin PF12 is automatically enabled if any of the three system timers specify their TACIx input as their clock source. When PF12 serves as UART RX, any of the system timers may be used for autobaud detection.

Internal Interfaces

Port control and GPIO registers are part of the system memory-mapped registers (MMRs). The addresses of the GPIO module MMRs appear in Appendix B. Core access to the GPIO configuration registers is through the system bus.

The PORTx_MUX registers control the muxing schemes of port F and port G.

The function enable registers (PORTF_FER and PORTG_FER) enable the peripheral functionality for each individual pin of a port.

Performance/Throughput

The PFx and PGx pins are synchronized to the system clock (SCLK). When configured as outputs, the GPIOs can transition once every system clock cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of port pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the signal is asserted on the pin and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.

Description of Operation

The operation of the general-purpose ports is described in the following sections.

Operation

The GPIO pins on port F and port G can be controlled individually by the function enable registers (PORTx_FER). With a control bit in these registers cleared, the peripheral function is fully decoupled from the pin. It functions as a GPIO pin only. To drive the pin in GPIO output mode, set the respective direction bit in the PORTxIO_DIR register. To make the pin a digital input or interrupt input, enable its input driver in the PORTxIO_INEN register.

By default all peripheral pins are configured as inputs after reset. port F and port G pins are in GPIO mode. However, GPIO input drivers are disabled to minimize power consumption and any need of external pulling resistors.

When the control bit in the function enable registers (PORTx_FER) is set, the pin is set to its peripheral functionality and is no longer controlled by the GPIO module. However, the GPIO module can still sense the state of

the pin. When using a particular peripheral interface, pins required for the peripheral must be individually enabled. Keep the related function enable bit cleared if a signal provided by the peripheral is not required by your application. This allows it to be used in GPIO mode.

General-Purpose I/O Modules

The processor supports 32 bidirectional or general-purpose I/O (GPIO) signals. These 32 GPIOs are managed by two different GPIO modules, which are functionally identical. One is associated with port F, and one is associated with port G. Port F and port G each consist of 16 GPIOs (PF15-0 and PG15-0), respectively.

Each GPIO can be individually configured as either an input or an output by using the GPIO direction registers (PORTxIO_DIR).

When configured as output, the GPIO data registers (PORTFIO, PORTGIO, and PORTHIO) can be directly written to specify the state of the GPIOs.

The GPIO direction registers are read-write registers with each bit position corresponding to a particular GPIO. A logic 1 configures a GPIO as an output, driving the state contained in the GPIO data register if the peripheral function is not enabled by the function enable registers. A logic 0 configures a GPIO as an input.

Note when using the GPIO as an input, the corresponding bit should also be set in the GPIO input enable register. Otherwise, changes at the input pins will not be recognized by the processor.

The GPIO input enable registers (PORTFIO_INEN and PORTGIO_INEN) are used to enable the input buffers on any GPIO that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx or PGx pin is not used in the system. By default, the input buffers are disabled.

Once the input driver of a GPIO pin is enabled, the GPIO is not allowed to operate as an output anymore. Never enable the input driver (by setting PORTxIO_INEN bits) and the output driver (by setting PORTxIO_DIR bits) for the same GPIO.

A write operation to any of the GPIO data registers sets the value of all GPIOs in this port that are configured as outputs. GPIOs configured as inputs ignore the written value. A read operation returns the state of the GPIOs defined as outputs and the sense of the inputs, based on the polarity and sensitivity settings, if their input buffers are enabled. Table 7-3 helps to interpret read values in GPIO mode, based on the settings of the PORTxIO_POLAR, PORTxIO_EDGE, and PORTxIO_BOTH registers.

Table 7-3. GPIO Value Register Pin Interpretation

POLAR	EDGE	BOTH	Effect of MMR Settings
0	0	X	Pin that is high reads as 1; pin that is low reads as 0
0	1	0	If rising edge occurred, pin reads as 1; otherwise, pin reads as 0
1	0	X	Pin that is low reads as 1; pin that is high reads as 0
1	1	0	If falling edge occurred, pin reads as 1; otherwise, pin reads as 0
X	1	1	If any edge occurred, pin reads as 1; otherwise, pin reads as 0

For GPIOs configured as edge-sensitive, a readback of 1 from one of these registers is sticky. That is, once it is set it remains set until cleared by user code. For level-sensitive GPIOs, the pin state is checked every cycle, so the readback value will change when the original level on the pin changes.

The state of the output is reflected on the associated pin only if the function enable bit in the PORTx_FER register is cleared.

Write operations to the GPIO data registers modify the state of all GPIOs of a port. In cases where only one or a few GPIOs need to be changed, the user may write to the GPIO set registers, PORTxIO_SET, the GPIO clear registers, PORTxIO_CLEAR, or to the GPIO toggle registers, PORTxIO_TOGGLE instead.

While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO set register can be used to set a single or a few bits only. No read-modify-write operations are required. The GPIO set registers are write-1-to-set registers. All 1s contained in the value written to a GPIO set register sets the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF0 is configured as an output. Writing 0x0001 to the GPIO set register drives a logic 1 on the PF0 pin without affecting the state of any other PFx pins. The GPIO set registers are typically also used to generate GPIO interrupts by software. Read operations from the GPIO set registers return the content of the GPIO data registers.

The GPIO clear registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO clear register can be used to clear individual bits only. No read-modify-write operations are required. The clear registers are write-1-to-clear registers. All 1s contained in the value written to the GPIO clear register clears the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF4 and PF5 are configured as outputs. Writing 0x0030 to the PORTFIO_CLEAR register drives a logic 0 on the PF4 and PF5 pins without affecting the state of any other PFx pins.

If an edge-sensitive pin generates an interrupt request, the service routine must acknowledge the request by clearing the respective GPIO latch. This is usually performed through the clear registers.

Read operations from the GPIO clear registers return the content of the GPIO data registers.

The GPIO toggle registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a toggle register can be used to toggle individual bits. No read-modify-write operations are required. The GPIO toggle registers are write-1-to-toggle registers. All 1s contained in the value written to a GPIO toggle register toggle the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PG1 is configured as an output. Writing 0x0002 to the PORTGIO_TOGGLE register changes the pin state (from logic 0 to logic 1, or from logic 1 to logic 0) on the PG1 pin without affecting the state of any other PGx pins. Read operations from the GPIO toggle registers return the content of the GPIO data registers.

The state of the GPIOs can be read through any of these data, set, clear, or toggle registers. However, the returned value reflects the state of the input pin only if the proper input enable bit in the PORTxIO_INEN register is set. Note that GPIOs can still sense the state of the pin when the function enable bits in the PORTx_FER registers are set.

Since function enable registers and GPIO input enable registers reset to zero, no external pull-ups or pull-downs are required on the unused pins of port F and port G.

GPIO Interrupt Processing

Each GPIO can be configured to generate an interrupt. The processor can sense up to 32 asynchronous off-chip signals, requesting interrupts through four interrupt channels. To make a pin function as an interrupt pin, the associated input enable bit in the PORTxIO_INEN register must be set. The function enable bit in the PORTx_FER register is typically cleared. Then, an interrupt request can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high and high to low). Input sensitivity is defined on a per-bit basis by the GPIO polarity registers (PORTFIO_POLAR and PORTGIO_POLAR), and the GPIO interrupt sensitivity registers (PORTFIO_EDGE and PORTGIO_EDGE). If configured for edge sensitivity, the

GPIO set on both edges registers (PORTFIO_BOTH and PORTGIO_BOTH) let the interrupt request generate on both edges.

The GPIO polarity registers are used to configure the polarity of the GPIO input source. To select active high or rising edge, set the bits in the GPIO polarity register to 0. To select active low or falling edge, set the bits in the GPIO polarity register to 1. This register has no effect on GPIOs that are defined as outputs. The contents of the GPIO polarity registers are cleared at reset, defaulting to active high polarity.

The GPIO interrupt sensitivity registers are used to configure each of the inputs as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. The GPIO interrupt sensitivity register has no effect on GPIOs that are defined as outputs. The contents of the GPIO interrupt sensitivity registers are cleared at reset, defaulting to level sensitivity.

The GPIO set on both edges registers are used to enable interrupt generation on both rising and falling edges. When a given GPIO has been set to edge-sensitive in the GPIO interrupt sensitivity register, setting the respective bit in the GPIO set on both edges register to both edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on GPIOs that are defined as level-sensitive or as outputs. See Table 7-3 on page 7-9 for information on how the GPIO set on both edges register interacts with the GPIO polarity and GPIO interrupt sensitivity registers.

When the GPIO's input drivers are enabled while the GPIO direction registers configure it as an output, software can trigger a GPIO interrupt by writing to the data/set/toggle registers. The interrupt service routine should clear the GPIO to acknowledge the request.

Each of the two GPIO modules provides two independent interrupt channels. Identical in functionality, these are called interrupt A and interrupt

B. Both interrupt channels have their own mask register which lets you assign the individual GPIOs to none, either, or both interrupt channels.

Since all mask registers reset to zero, none of the GPIOs is assigned any interrupt by default. Each GPIO represents a bit in each of these registers. Setting a bit means enabling the interrupt on this channel.

Interrupt A and interrupt B operate independently. For example, writing 1 to a bit in the mask interrupt A register does not affect interrupt channel B. This facility allows GPIOs to generate GPIO interrupt A, GPIO interrupt B, both GPIO interrupts A and B, or neither.

A GPIO interrupt is generated by a logical OR of all unmasked GPIOs for that interrupt. For example, if PF0 and PF1 are both unmasked for GPIO interrupt channel A, GPIO interrupt A will be generated when triggered by PF0 or PF1. The interrupt service routine must evaluate the GPIO data register to determine the signaling interrupt source. Figure 7-1 illustrates the interrupt flow of any GPIO module's interrupt A channel.

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate bit in the GPIO clear register.

At reset, all interrupts are masked and disabled.

Similarly to the GPIOs themselves, the mask register can either be written through the GPIO mask data registers (PORTxIO_MASKA, PORTxIO_MASKB) or be controlled by the mask A/mask B set, clear and toggle registers.

The GPIO mask interrupt set registers (PORTxIO_MASKA_SET, PORTxIO_MASKB_SET) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt set register can be used to set a single or a few bits only. No read-modify-write operations are required.

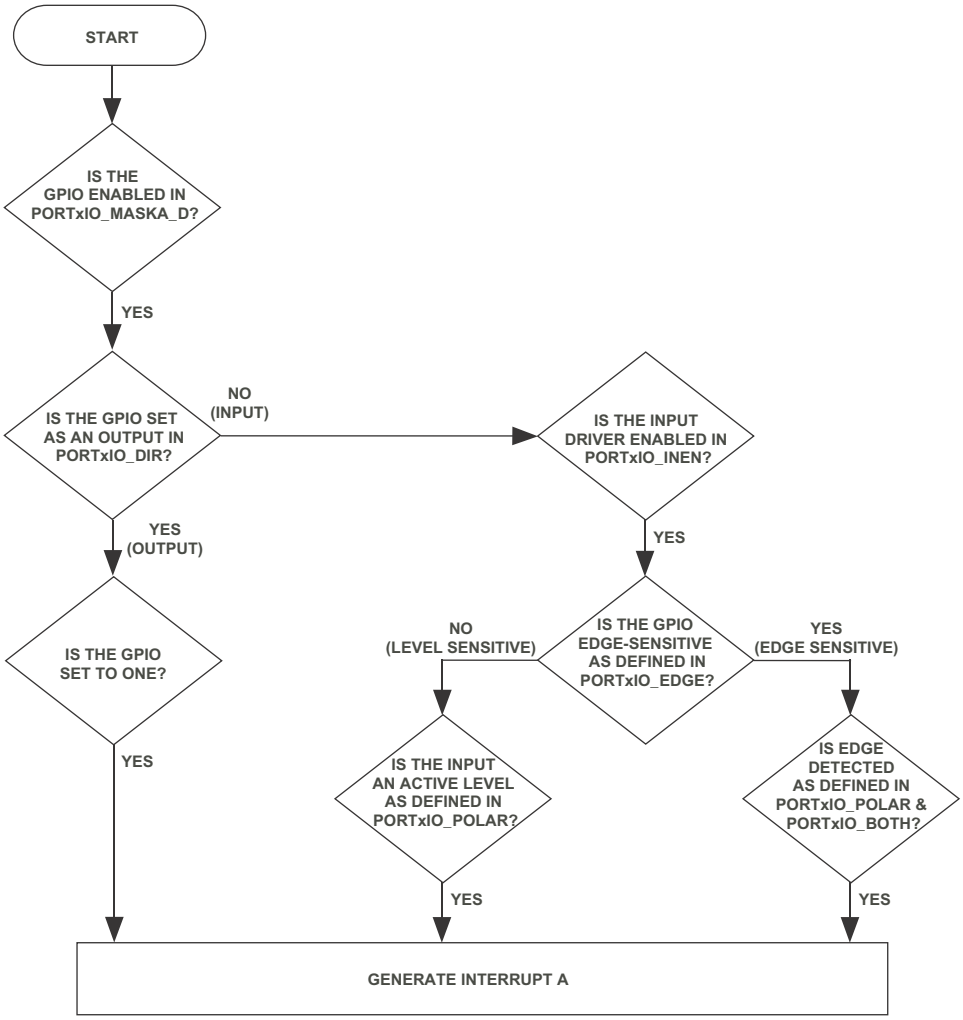

START
IS THE GPIO ENABLED IN PORTxIO_MASKA_D?
YES
IS THE GPIO SET AS AN OUTPUT IN PORTxIO_DIR?
NO (INPUT)
IS THE INPUT DRIVER ENABLED IN PORTxIO_INEN?
YES (OUTPUT)
YES
IS THE GPIO SET TO ONE?
NO (LEVEL SENSITIVE)
IS THE GPIO EDGE-SENSITIVE AS DEFINED IN PORTxIO_EDGE?
YES (EDGE SENSITIVE)
YES
IS THE INPUT AN ACTIVE LEVEL AS DEFINED IN PORTxIO_POLAR?
IS EDGE DETECTED AS DEFINED IN PORTxIO_POLAR & PORTxIO_BOTH?
YES
YES
GENERATE INTERRUPT A

Figure 7-1. GPIO Interrupt Generation Flow for Interrupt Channel A

The mask interrupt set registers are write-1-to-set registers. All ones contained in the value written to the mask interrupt set register set the

respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit enables the interrupt for the respective GPIO.

The GPIO mask interrupt clear registers (PORTxIO_MASKA_CLEAR, PORTxIO_MASKB_CLEAR) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to the mask interrupt clear register can be used to clear a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt clear registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt clear register clear the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit disables the interrupt for the respective GPIO.

The GPIO mask interrupt toggle registers (PORTxIO_MASKA_TOGGLE, PORTxIO_MASKB_TOGGLE) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt toggle register can be used to toggle a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt toggle registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt toggle register toggle the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit toggles the interrupt for the respective GPIO.

Figure 7-1 illustrates the interrupt flow of any GPIO module's interrupt A channel. The interrupt B channel behaves identically.

All GPIOs assigned to the same interrupt channel are OR'ed. If multiple GPIOs are assigned to the same interrupt channel, it is up to the interrupt service routine to evaluate the GPIO data registers to determine the signaling interrupt source.

Figure 7-2 shows the mapping of the four GPIO interrupt channels of port F and port G.

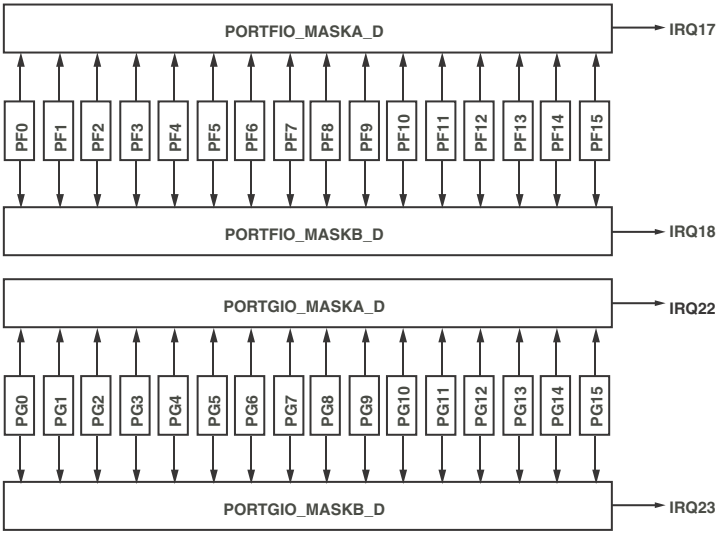


Figure 7-2. GPIO Interrupt Channels

Programming Model

Figure 7-3 and Figure 7-4 show the programming model for the general-purpose ports.

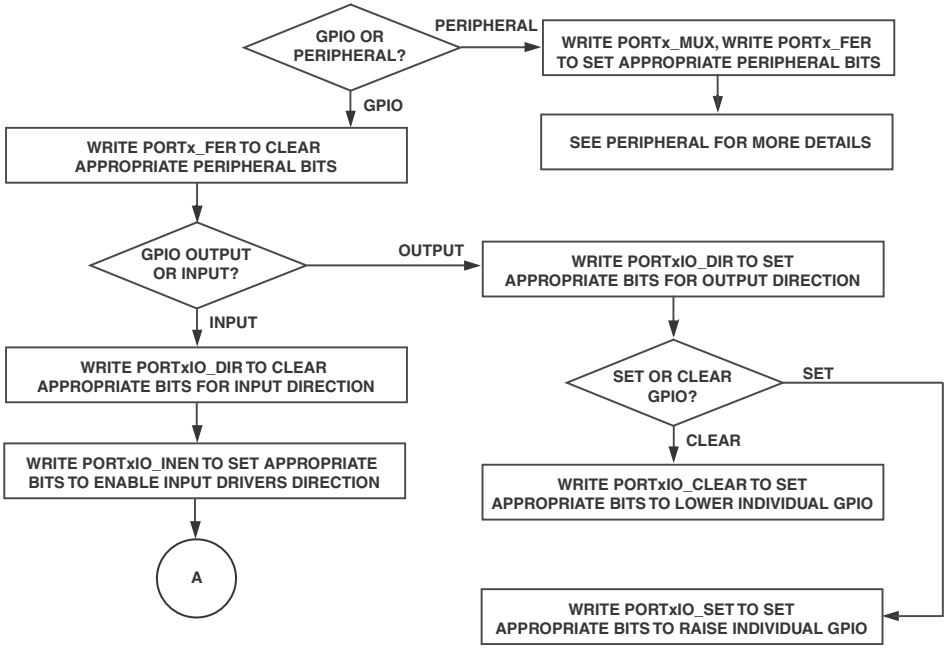

GPIO OR PERIPHERAL?
PERIPHERAL
WRITE PORTx_MUX, WRITE PORTx_FER TO SET APPROPRIATE PERIPHERAL BITS
SEE PERIPHERAL FOR MORE DETAILS
GPIO
WRITE PORTx_FER TO CLEAR APPROPRIATE PERIPHERAL BITS
GPIO OUTPUT OR INPUT?
OUTPUT
WRITE PORTxIO_DIR TO SET APPROPRIATE BITS FOR OUTPUT DIRECTION
INPUT
WRITE PORTxIO_DIR TO CLEAR APPROPRIATE BITS FOR INPUT DIRECTION
SET OR CLEAR GPIO?
SET
CLEAR
WRITE PORTxIO_INEN TO SET APPROPRIATE BITS TO ENABLE INPUT DRIVERS DIRECTION
WRITE PORTxIO_CLEAR TO SET APPROPRIATE BITS TO LOWER INDIVIDUAL GPIO
A
WRITE PORTxIO_SET TO SET APPROPRIATE BITS TO RAISE INDIVIDUAL GPIO

Figure 7-3. GPIO Flow Chart (Part 1 of 2)

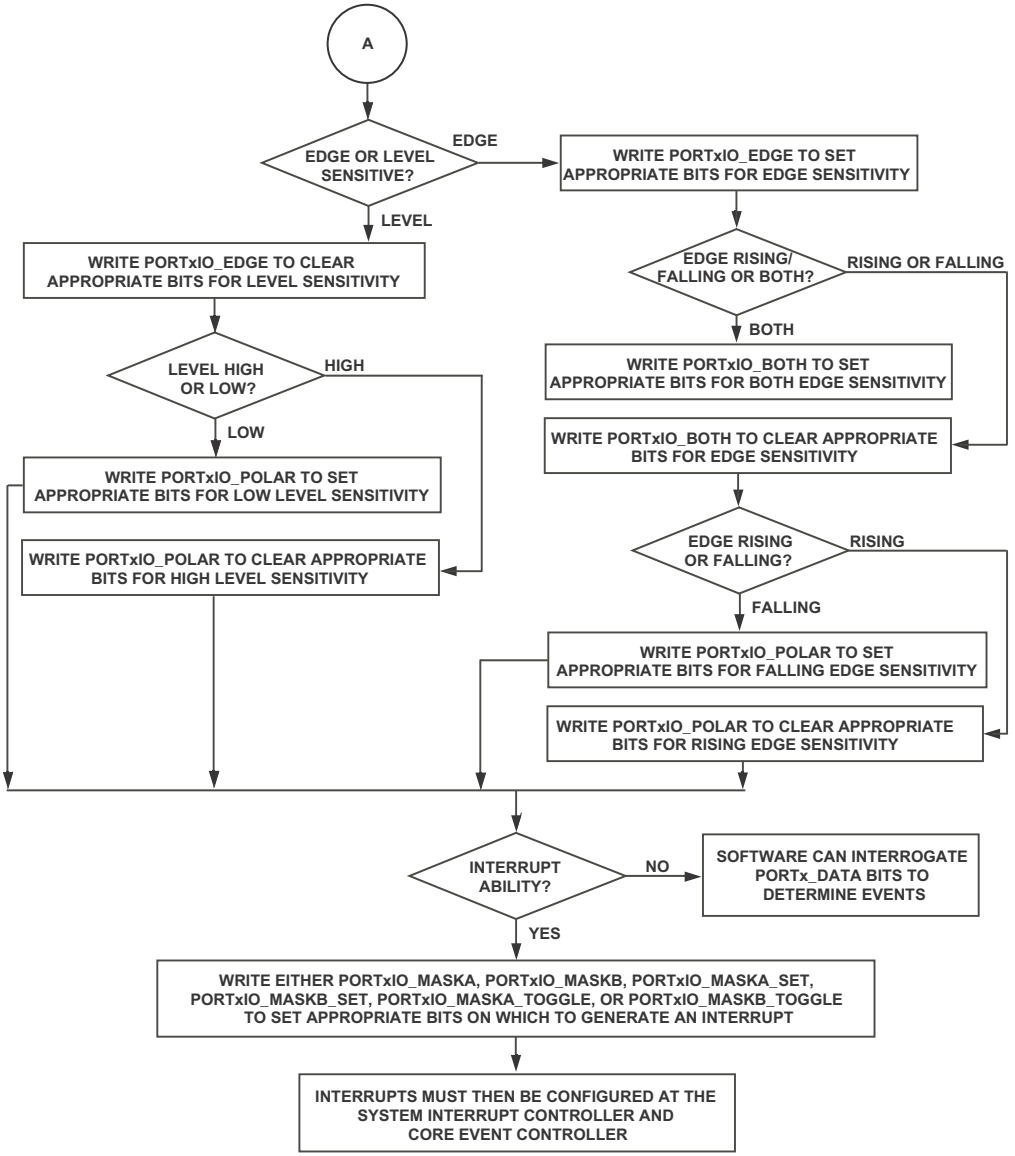

A
EDGE OR LEVEL SENSITIVE?
EDGE
LEVEL
WRITE PORTxIO_EDGE TO SET APPROPRIATE BITS FOR EDGE SENSITIVITY
WRITE PORTxIO_EDGE TO CLEAR APPROPRIATE BITS FOR LEVEL SENSITIVITY
EDGE RISING/ FALLING OR BOTH?
RISING OR FALLING
BOTH
LEVEL HIGH OR LOW?
HIGH
LOW
WRITE PORTxIO_BOTH TO SET APPROPRIATE BITS FOR BOTH EDGE SENSITIVITY
WRITE PORTxIO_BOTH TO CLEAR APPROPRIATE BITS FOR EDGE SENSITIVITY
WRITE PORTxIO_POLAR TO SET APPROPRIATE BITS FOR LOW LEVEL SENSITIVITY
WRITE PORTxIO_POLAR TO CLEAR APPROPRIATE BITS FOR HIGH LEVEL SENSITIVITY
EDGE RISING OR FALLING?
RISING
FALLING
WRITE PORTxIO_POLAR TO SET APPROPRIATE BITS FOR FALLING EDGE SENSITIVITY
WRITE PORTxIO_POLAR TO CLEAR APPROPRIATE BITS FOR RISING EDGE SENSITIVITY
INTERRUPT ABILITY?
NO
SOFTWARE CAN INTERROGATE PORTx_DATA BITS TO DETERMINE EVENTS
YES
WRITE EITHER PORTxIO_MASKA, PORTxIO_MASKB, PORTxIO_MASKA_SET, PORTxIO_MASKB_SET, PORTxIO_MASKA_TOGGLE, OR PORTxIO_MASKB_TOGGLE TO SET APPROPRIATE BITS ON WHICH TO GENERATE AN INTERRUPT
INTERRUPTS MUST THEN BE CONFIGURED AT THE SYSTEM INTERRUPT CONTROLLER AND CORE EVENT CONTROLLER

Figure 7-4. GPIO Flow Chart (Part 2 of 2)

GPIO Schmitt Trigger Control

The ADSP-BF59x contains additional registers controlling the hysteresis (via Schmitt triggering) for Port F and Port G. These are also included for several pins and group of pins other than GPIOs. Figure 7-5 on page 7-19 to Figure 7-6 on page 7-20 show the bit descriptions of these registers.

PORTx Pad Control Registers

These registers configure hysteresis for the PORTx inputs. For each controlled group of pins, b#0 will disable Schmitt triggering (hysteresis), while b#1 will enable it.

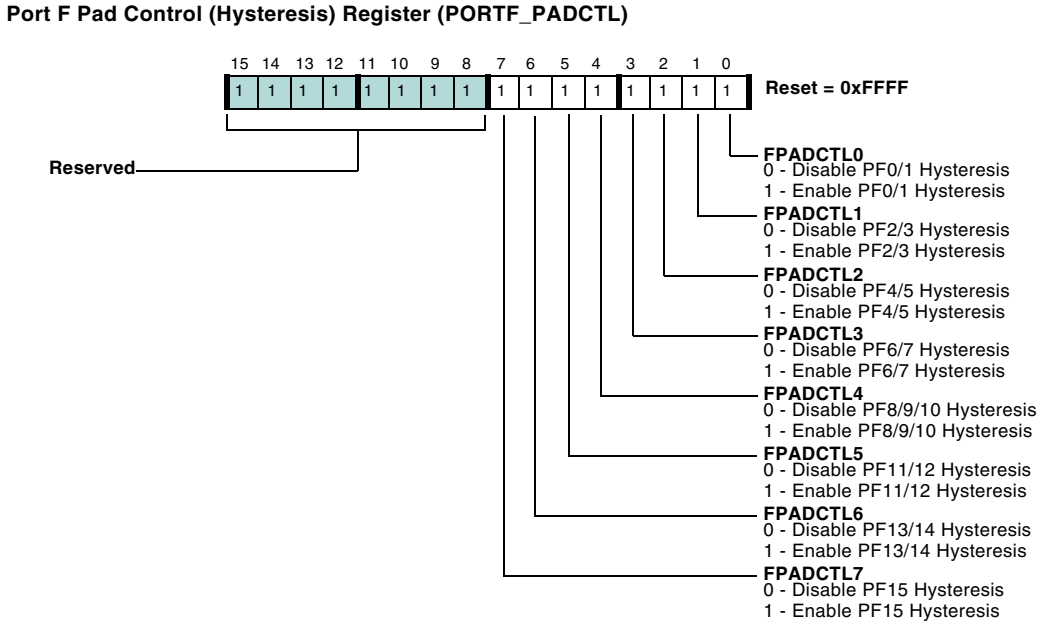

Port F Pad Control (Hysteresis) Register (PORTF_PADCTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Reset = 0xFFFF
Reserved
FPADCTL0
0 - Disable PF0/1 Hysteresis
1 - Enable PF0/1 Hysteresis
FPADCTL1
0 - Disable PF2/3 Hysteresis
1 - Enable PF2/3 Hysteresis
FPADCTL2
0 - Disable PF4/5 Hysteresis
1 - Enable PF4/5 Hysteresis
FPADCTL3
0 - Disable PF6/7 Hysteresis
1 - Enable PF6/7 Hysteresis
FPADCTL4
0 - Disable PF8/9/10 Hysteresis
1 - Enable PF8/9/10 Hysteresis
FPADCTL5
0 - Disable PF11/12 Hysteresis
1 - Enable PF11/12 Hysteresis
FPADCTL6
0 - Disable PF13/14 Hysteresis
1 - Enable PF13/14 Hysteresis
FPADCTL7
0 - Disable PF15 Hysteresis
1 - Enable PF15 Hysteresis

Figure 7-5. Port F Pad Control (Hysteresis) Register

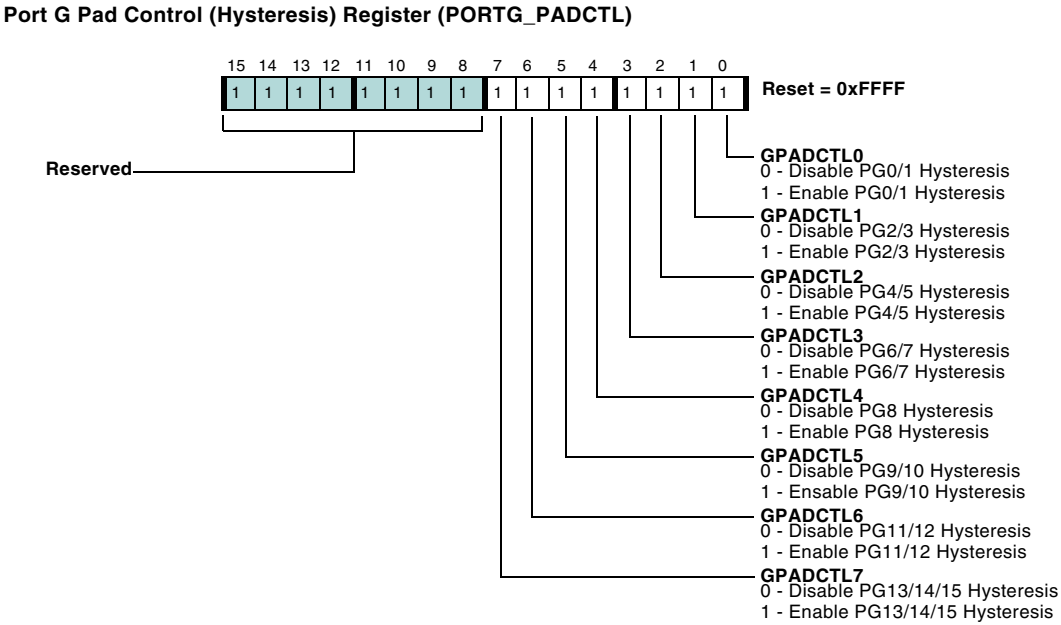
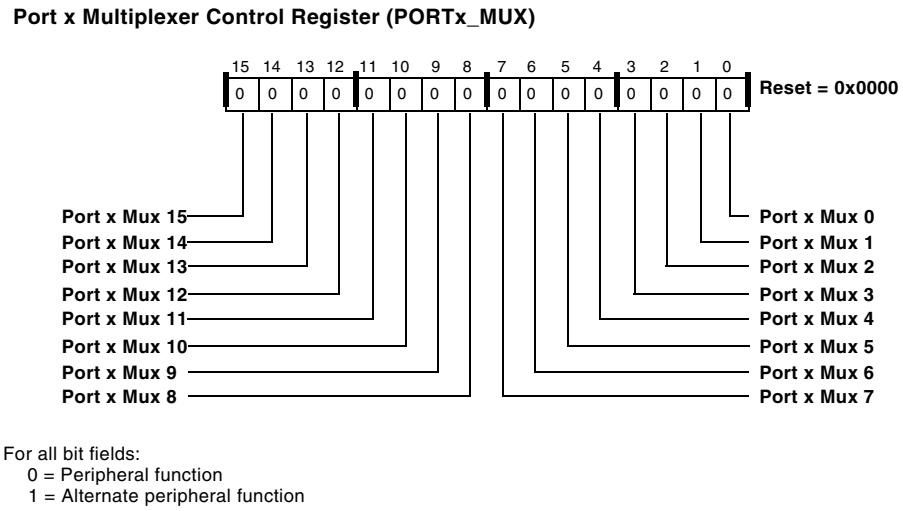

Port G Pad Control (Hysteresis) Register (PORTG_PADCTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Reset = 0xFFFF
Reserved
GPADCTL0
0 - Disable PG0/1 Hysteresis
1 - Enable PG0/1 Hysteresis
GPADCTL1
0 - Disable PG2/3 Hysteresis
1 - Enable PG2/3 Hysteresis
GPADCTL2
0 - Disable PG4/5 Hysteresis
1 - Enable PG4/5 Hysteresis
GPADCTL3
0 - Disable PG6/7 Hysteresis
1 - Enable PG6/7 Hysteresis
GPADCTL4
0 - Disable PG8 Hysteresis
1 - Enable PG8 Hysteresis
GPADCTL5
0 - Disable PG9/10 Hysteresis
1 - Ensable PG9/10 Hysteresis
GPADCTL6
0 - Disable PG11/12 Hysteresis
1 - Enable PG11/12 Hysteresis
GPADCTL7
0 - Disable PG13/14/15 Hysteresis
1 - Enable PG13/14/15 Hysteresis

Figure 7-6. Port G Pad Control (Hysteresis) Register

Memory-Mapped GPIO Registers

The GPIO registers are part of the system memory-mapped registers (MMRs). Figure 7-7 through Figure 7-25 on page 7-33 illustrate the GPIO registers. The addresses of the programmable flag MMRs appear in "System MMR Assignments" on page A-1.

Port Multiplexer Control Register (PORTx_MUX)


Port x Multiplexer Control Register (PORTx_MUX)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Port x Mux 15
Port x Mux 14
Port x Mux 13
Port x Mux 12
Port x Mux 11
Port x Mux 10
Port x Mux 9
Port x Mux 8
Port x Mux 0
Port x Mux 1
Port x Mux 2
Port x Mux 3
Port x Mux 4
Port x Mux 5
Port x Mux 6
Port x Mux 7
For all bit fields:
0 = Peripheral function
1 = Alternate peripheral function

Refer to Table 7-1 on page 7-3 to Table 7-2 on page 7-4 for reserved bits in the PORTx_MUX register.

Figure 7-7. Port Multiplexer Control Register

Function Enable Registers (PORTx_FER)

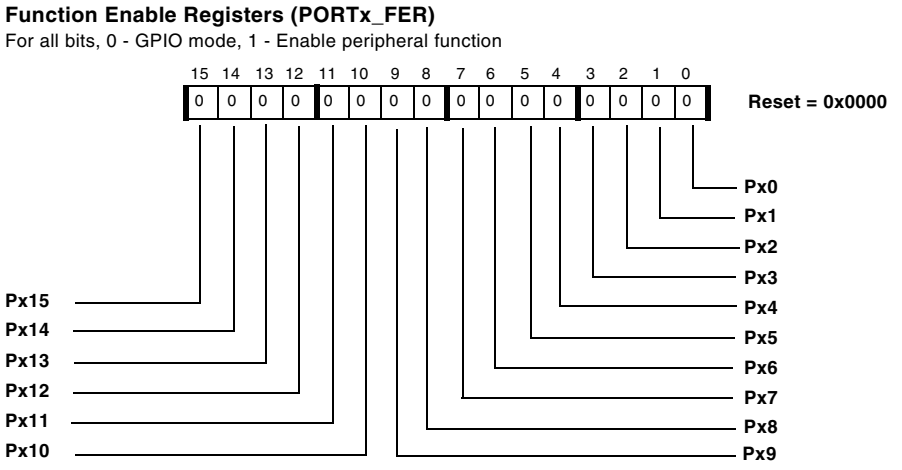


Figure 7-8. Function Enable Registers

GPIO Direction Registers (PORTxIO_DIR)

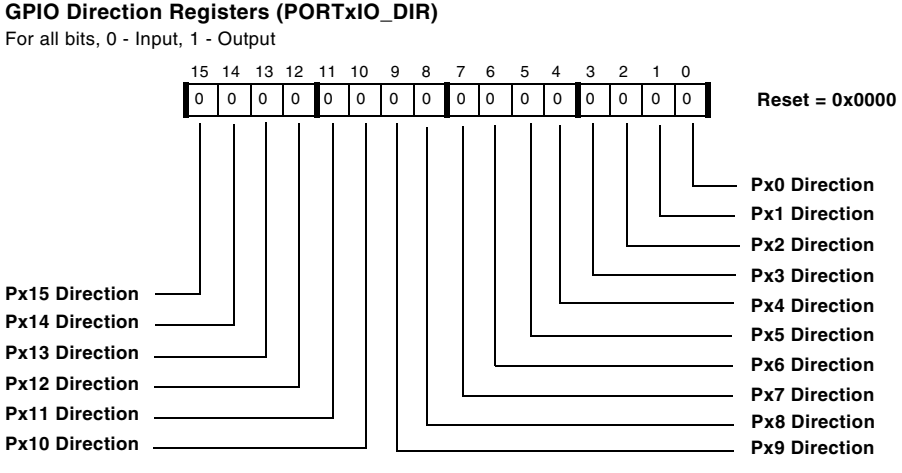


Figure 7-9. GPIO Direction Registers

GPIO Input Enable Registers (PORTxIO_INEN)

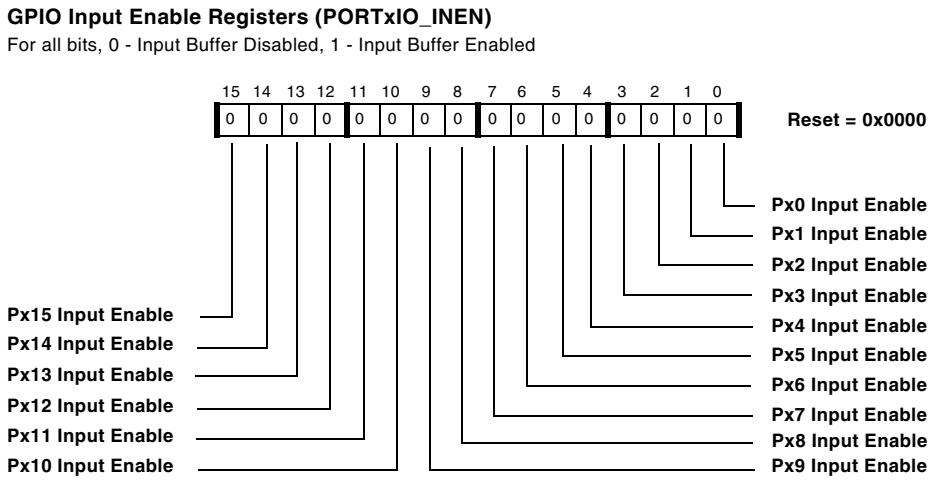

GPIO Input Enable Registers (PORTxIO_INEN)
For all bits, 0 - Input Buffer Disabled, 1 - Input Buffer Enabled
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Px0 Input Enable
Px1 Input Enable
Px2 Input Enable
Px3 Input Enable
Px4 Input Enable
Px5 Input Enable
Px6 Input Enable
Px7 Input Enable
Px8 Input Enable
Px9 Input Enable
Px10 Input Enable
Px11 Input Enable
Px12 Input Enable
Px13 Input Enable
Px14 Input Enable
Px15 Input Enable

Figure 7-10. GPIO Input Enable Registers

GPIO Data Registers (PORTxIO)

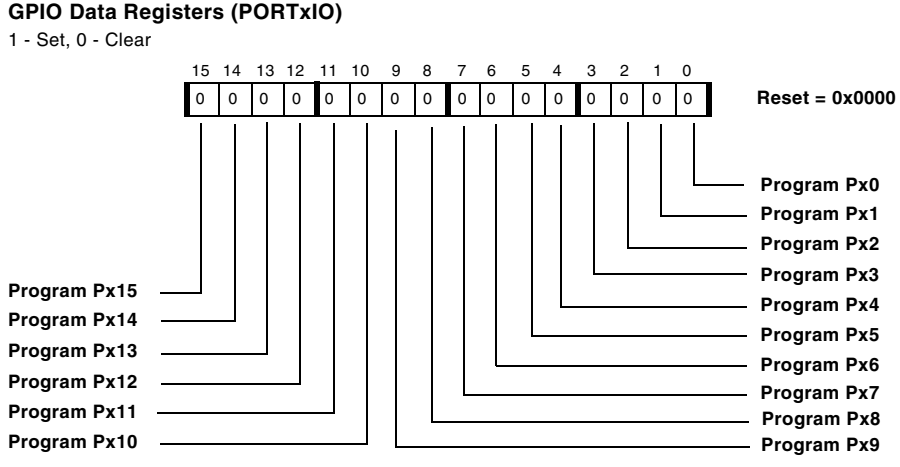


Figure 7-11. GPIO Data Registers

GPIO Set Registers (PORTxIO_SET)

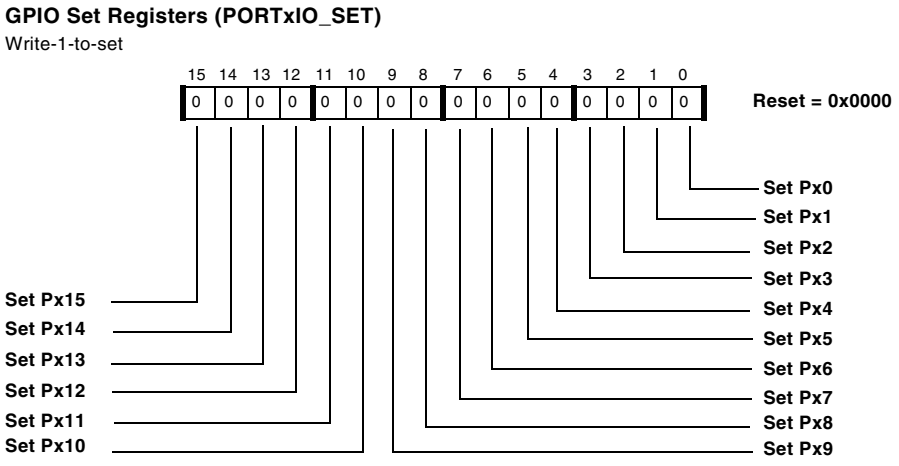


Figure 7-12. GPIO Set Registers

GPIO Clear Registers (PORTxIO_CLEAR)

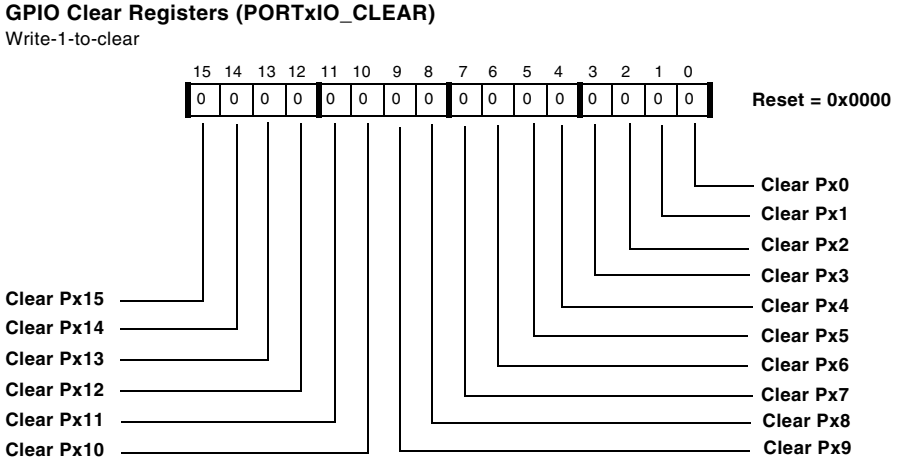


Figure 7-13. GPIO Clear Registers

GPIO Toggle Registers (PORTxIO_TOGGLE)

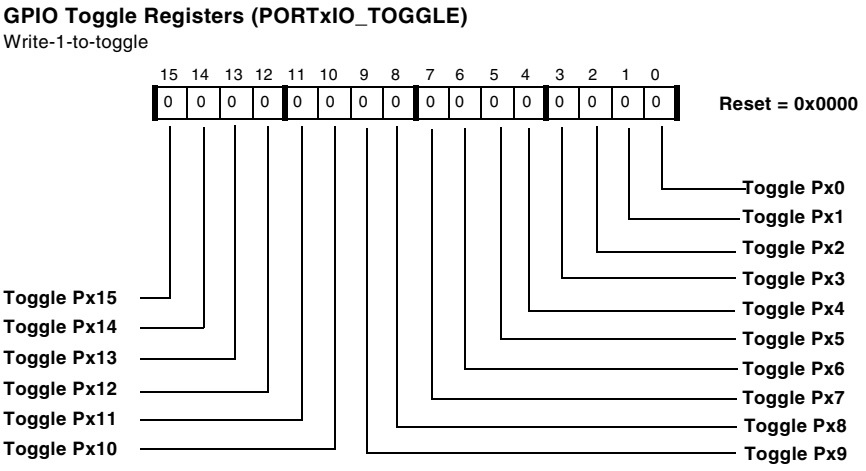


Figure 7-14. GPIO Toggle Registers

GPIO Polarity Registers (PORTxIO_POLAR)

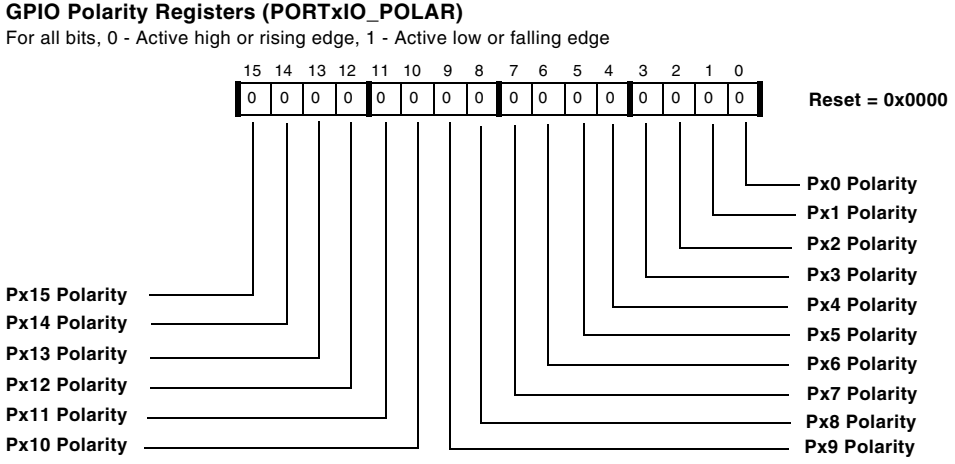

GPIO Polarity Registers (PORTxIO_POLAR)
For all bits, 0 - Active high or rising edge, 1 - Active low or falling edge
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Px0 Polarity
Px1 Polarity
Px2 Polarity
Px3 Polarity
Px4 Polarity
Px5 Polarity
Px6 Polarity
Px7 Polarity
Px8 Polarity
Px9 Polarity
Px10 Polarity
Px11 Polarity
Px12 Polarity
Px13 Polarity
Px14 Polarity
Px15 Polarity

Figure 7-15. GPIO Polarity Registers

Interrupt Sensitivity Registers (PORTxIO_EDGE)

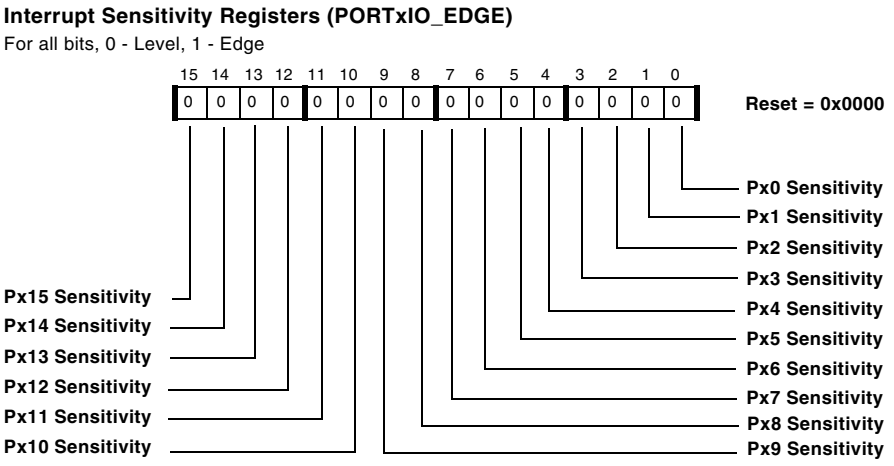


Figure 7-16. Interrupt Sensitivity Registers

GPIO Set on Both Edges Registers (PORTxIO_BOTH)

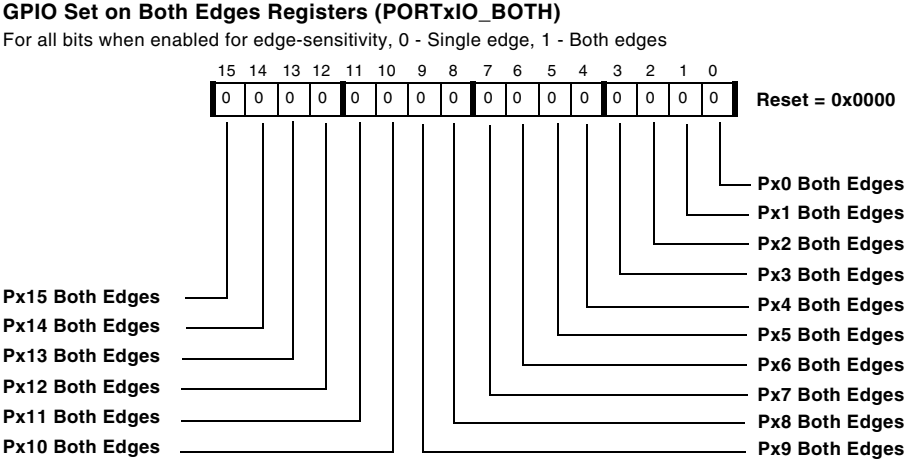

GPIO Set on Both Edges Registers (PORTxIO_BOTH)
For all bits when enabled for edge-sensitivity, 0 - Single edge, 1 - Both edges
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Px0 Both Edges
Px1 Both Edges
Px2 Both Edges
Px3 Both Edges
Px4 Both Edges
Px5 Both Edges
Px6 Both Edges
Px7 Both Edges
Px8 Both Edges
Px9 Both Edges
Px15 Both Edges
Px14 Both Edges
Px13 Both Edges
Px12 Both Edges
Px11 Both Edges
Px10 Both Edges

Figure 7-17. GPIO Set on Both Edges Registers

GPIO Mask Interrupt Registers (PORTxIO_MASKA/B)

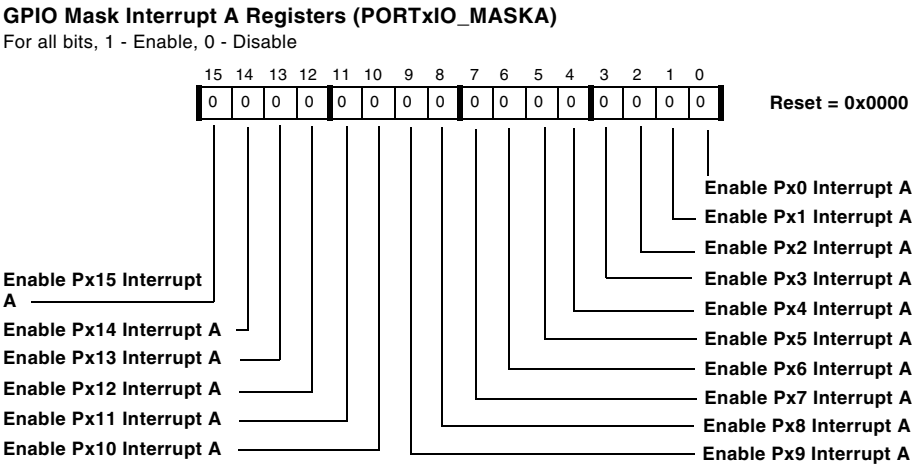

GPIO Mask Interrupt A Registers (PORTxIO_MASKA)
For all bits, 1 - Enable, 0 - Disable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Enable Px0 Interrupt A
Enable Px1 Interrupt A
Enable Px2 Interrupt A
Enable Px3 Interrupt A
Enable Px4 Interrupt A
Enable Px5 Interrupt A
Enable Px6 Interrupt A
Enable Px7 Interrupt A
Enable Px8 Interrupt A
Enable Px9 Interrupt A
Enable Px10 Interrupt A
Enable Px11 Interrupt A
Enable Px12 Interrupt A
Enable Px13 Interrupt A
Enable Px14 Interrupt A
Enable Px15 Interrupt A

Figure 7-18. GPIO Mask Interrupt A Registers

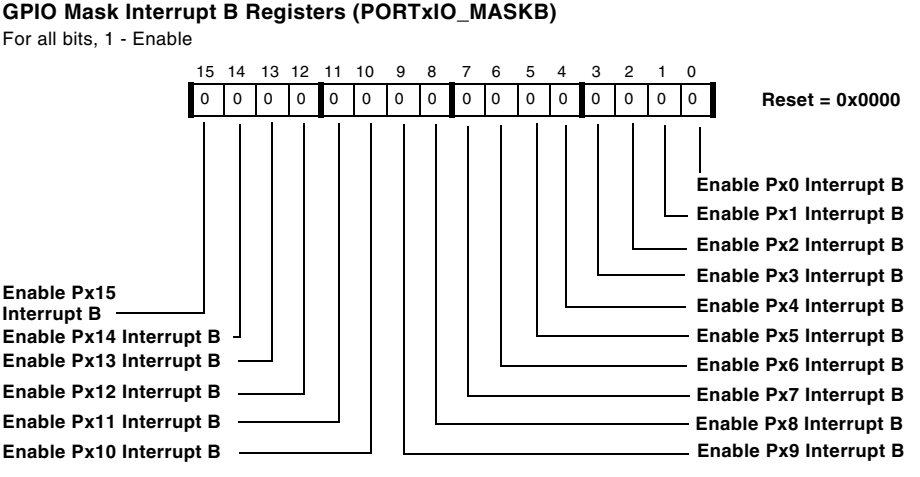

GPIO Mask Interrupt B Registers (PORTxIO_MASKB)
For all bits, 1 - Enable
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Enable Px0 Interrupt B
Enable Px1 Interrupt B
Enable Px2 Interrupt B
Enable Px3 Interrupt B
Enable Px4 Interrupt B
Enable Px5 Interrupt B
Enable Px6 Interrupt B
Enable Px7 Interrupt B
Enable Px8 Interrupt B
Enable Px9 Interrupt B
Enable Px10 Interrupt B
Enable Px11 Interrupt B
Enable Px12 Interrupt B
Enable Px13 Interrupt B
Enable Px14 Interrupt B
Enable Px15 Interrupt B

Figure 7-19. GPIO Mask Interrupt B Registers

GPIO Mask Interrupt Set Registers (PORTxIO_MASKA/B_SET)

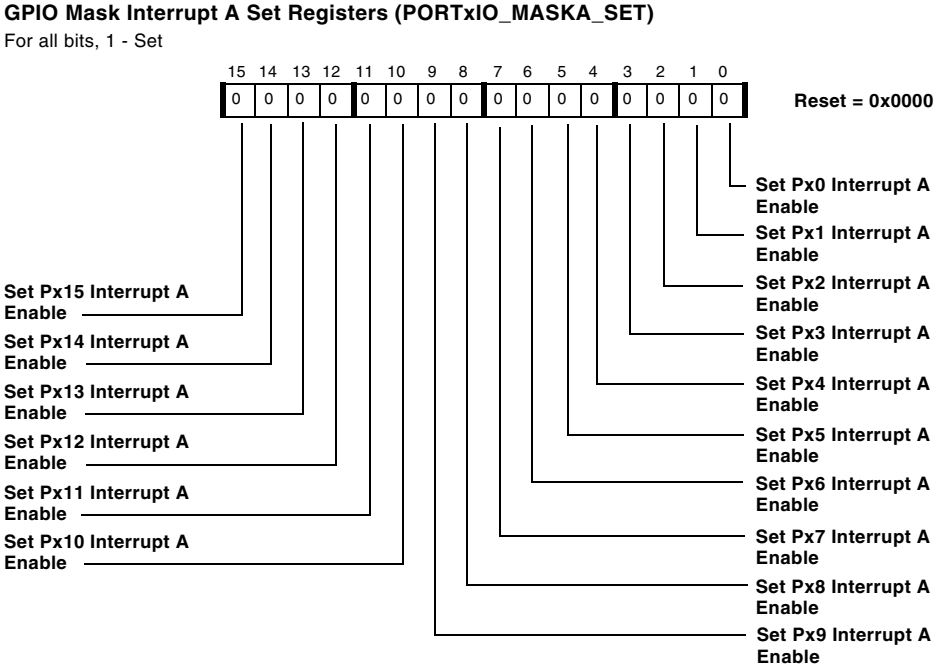

GPIO Mask Interrupt A Set Registers (PORTxIO_MASKA_SET)
For all bits, 1 - Set
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Set Px0 Interrupt A Enable
Set Px1 Interrupt A Enable
Set Px2 Interrupt A Enable
Set Px3 Interrupt A Enable
Set Px4 Interrupt A Enable
Set Px5 Interrupt A Enable
Set Px6 Interrupt A Enable
Set Px7 Interrupt A Enable
Set Px8 Interrupt A Enable
Set Px9 Interrupt A Enable
Set Px10 Interrupt A Enable
Set Px11 Interrupt A Enable
Set Px12 Interrupt A Enable
Set Px13 Interrupt A Enable
Set Px14 Interrupt A Enable
Set Px15 Interrupt A Enable

Figure 7-20. GPIO Mask Interrupt A Set Registers

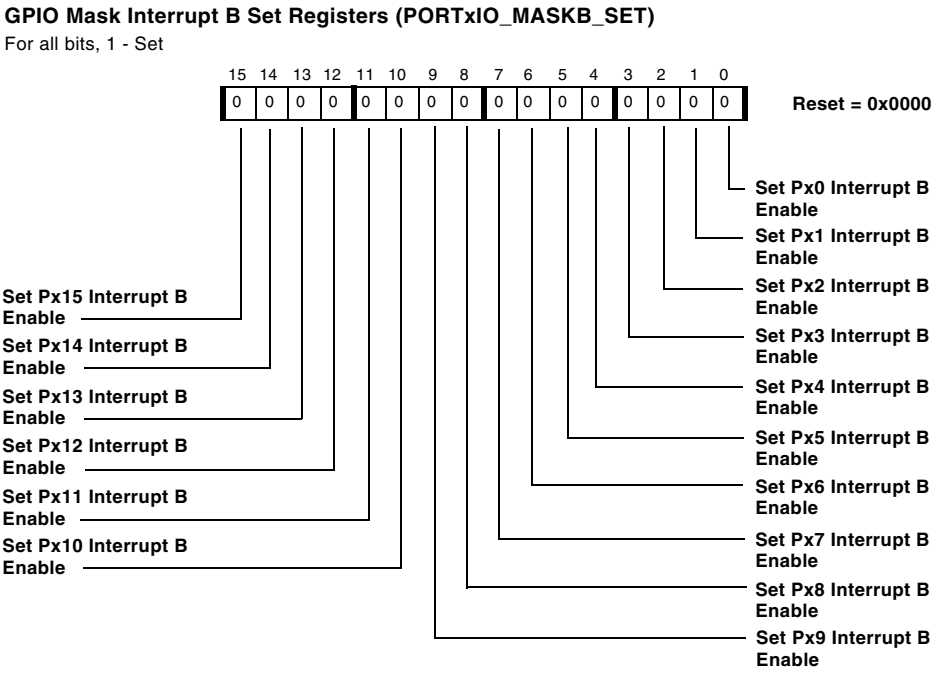

GPIO Mask Interrupt B Set Registers (PORTxIO_MASKB_SET)
For all bits, 1 - Set
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Set Px0 Interrupt B Enable
Set Px1 Interrupt B Enable
Set Px2 Interrupt B Enable
Set Px3 Interrupt B Enable
Set Px4 Interrupt B Enable
Set Px5 Interrupt B Enable
Set Px6 Interrupt B Enable
Set Px7 Interrupt B Enable
Set Px8 Interrupt B Enable
Set Px9 Interrupt B Enable
Set Px10 Interrupt B Enable
Set Px11 Interrupt B Enable
Set Px12 Interrupt B Enable
Set Px13 Interrupt B Enable
Set Px14 Interrupt B Enable
Set Px15 Interrupt B Enable

Figure 7-21. GPIO Mask Interrupt B Set Registers

GPIO Mask Interrupt Clear Registers (PORTxIO_MASKA/B_CLEAR)

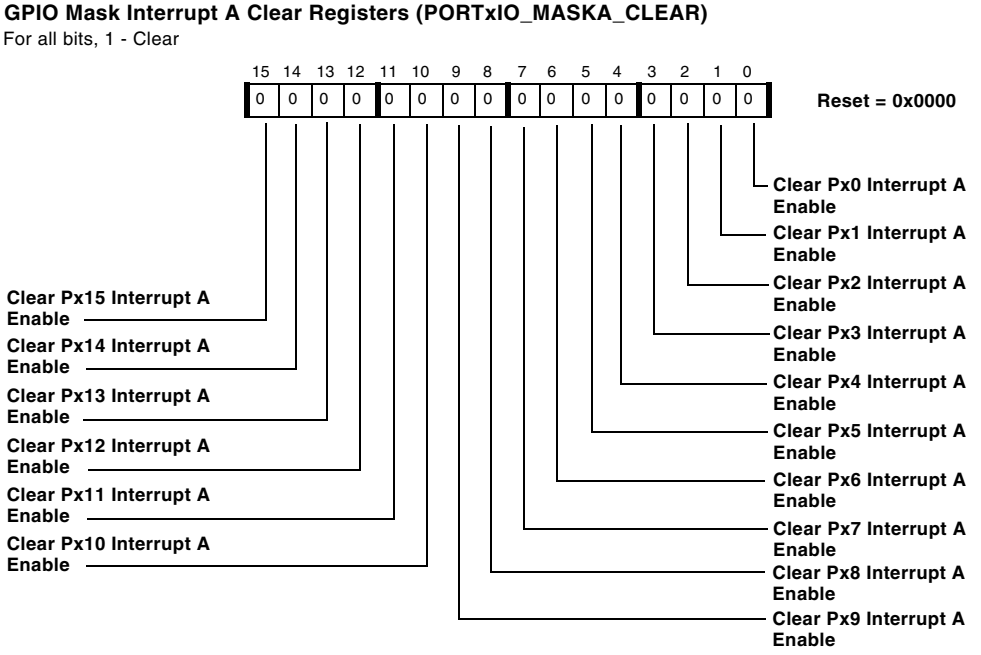

GPIO Mask Interrupt A Clear Registers (PORTxIO_MASKA_CLEAR)
For all bits, 1 - Clear
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Clear Px0 Interrupt A Enable
Clear Px1 Interrupt A Enable
Clear Px2 Interrupt A Enable
Clear Px3 Interrupt A Enable
Clear Px4 Interrupt A Enable
Clear Px5 Interrupt A Enable
Clear Px6 Interrupt A Enable
Clear Px7 Interrupt A Enable
Clear Px8 Interrupt A Enable
Clear Px9 Interrupt A Enable
Clear Px10 Interrupt A Enable
Clear Px11 Interrupt A Enable
Clear Px12 Interrupt A Enable
Clear Px13 Interrupt A Enable
Clear Px14 Interrupt A Enable
Clear Px15 Interrupt A Enable

Figure 7-22. GPIO Mask Interrupt A Clear Registers

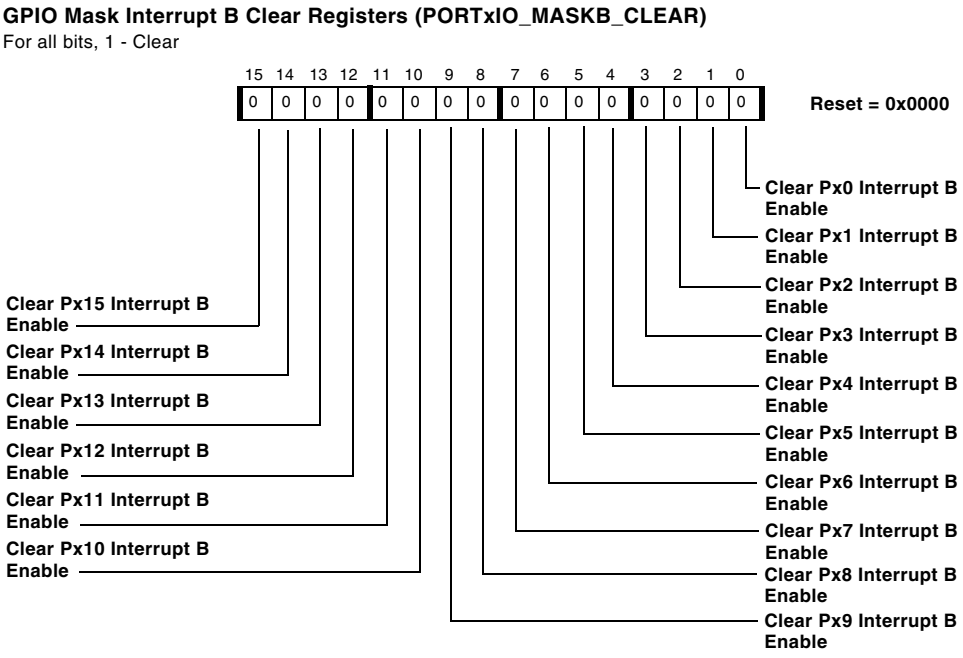

GPIO Mask Interrupt B Clear Registers (PORTxIO_MASKB_CLEAR)
For all bits, 1 - Clear
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Clear Px0 Interrupt B Enable
Clear Px1 Interrupt B Enable
Clear Px2 Interrupt B Enable
Clear Px3 Interrupt B Enable
Clear Px4 Interrupt B Enable
Clear Px5 Interrupt B Enable
Clear Px6 Interrupt B Enable
Clear Px7 Interrupt B Enable
Clear Px8 Interrupt B Enable
Clear Px9 Interrupt B Enable
Clear Px10 Interrupt B Enable
Clear Px11 Interrupt B Enable
Clear Px12 Interrupt B Enable
Clear Px13 Interrupt B Enable
Clear Px14 Interrupt B Enable
Clear Px15 Interrupt B Enable

Figure 7-23. GPIO Mask Interrupt B Clear Registers

GPIO Mask Interrupt Toggle Registers (PORTxIO_MASKA/B_TOGGLE)

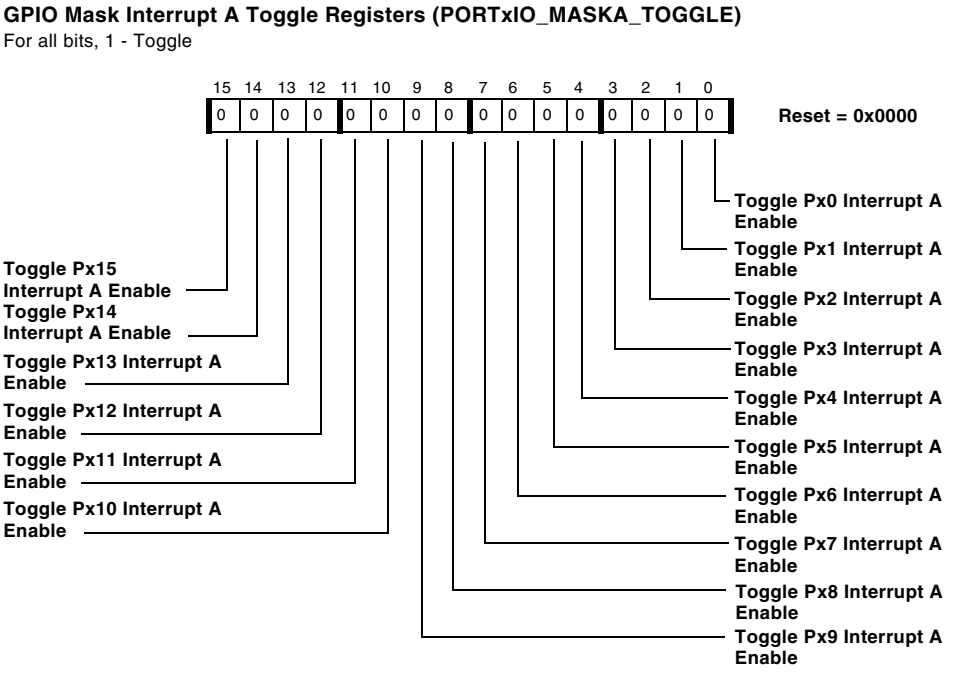

GPIO Mask Interrupt A Toggle Registers (PORTxIO_MASKA_TOGGLE)
For all bits, 1 - Toggle
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Toggle Px15 Interrupt A Enable
Toggle Px14 Interrupt A Enable
Toggle Px13 Interrupt A Enable
Toggle Px12 Interrupt A Enable
Toggle Px11 Interrupt A Enable
Toggle Px10 Interrupt A Enable
Toggle Px0 Interrupt A Enable
Toggle Px1 Interrupt A Enable
Toggle Px2 Interrupt A Enable
Toggle Px3 Interrupt A Enable
Toggle Px4 Interrupt A Enable
Toggle Px5 Interrupt A Enable
Toggle Px6 Interrupt A Enable
Toggle Px7 Interrupt A Enable
Toggle Px8 Interrupt A Enable
Toggle Px9 Interrupt A Enable

Figure 7-24. GPIO Mask Interrupt A Toggle Registers

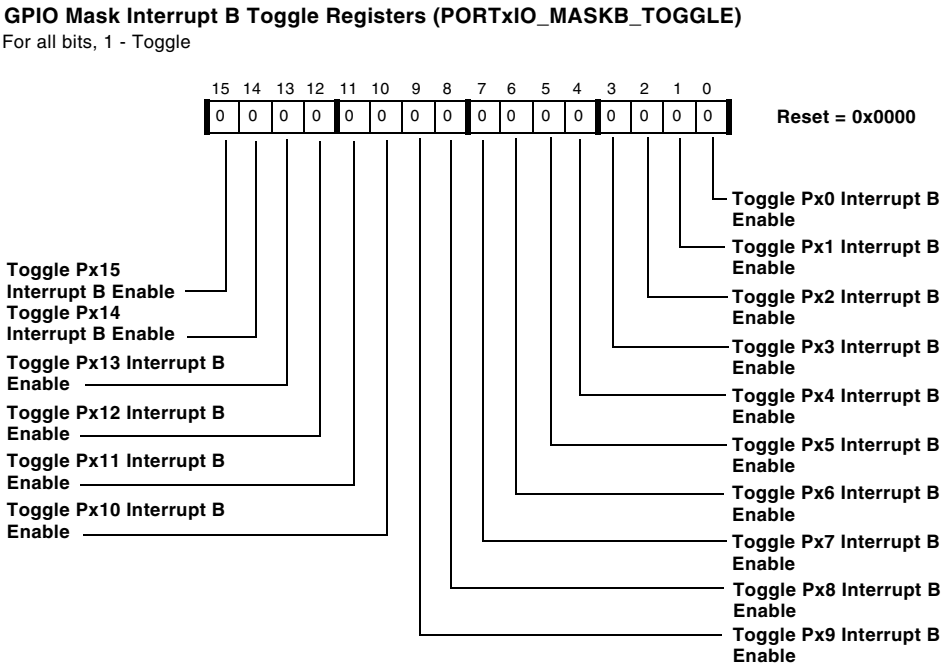

GPIO Mask Interrupt B Toggle Registers (PORTxIO_MASKB_TOGGLE)
For all bits, 1 - Toggle
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Toggle Px15 Interrupt B Enable
Toggle Px14 Interrupt B Enable
Toggle Px13 Interrupt B Enable
Toggle Px12 Interrupt B Enable
Toggle Px11 Interrupt B Enable
Toggle Px10 Interrupt B Enable
Toggle Px0 Interrupt B Enable
Toggle Px1 Interrupt B Enable
Toggle Px2 Interrupt B Enable
Toggle Px3 Interrupt B Enable
Toggle Px4 Interrupt B Enable
Toggle Px5 Interrupt B Enable
Toggle Px6 Interrupt B Enable
Toggle Px7 Interrupt B Enable
Toggle Px8 Interrupt B Enable
Toggle Px9 Interrupt B Enable

Figure 7-25. GPIO Mask Interrupt B Toggle Registers

Programming Examples

Listing 7-1 provides examples for using the general-purpose ports.

Listing 7-1. General-Purpose Ports

```
/* set port f function enable register to GPIO (not peripheral)
*/
p0.l = lo(PORTF_FER);
p0.h = hi(PORTF_FER);
R0.h = 0x0000;
r0.l = 0x0000;
w[p0] = r0;
```

```
/* set port f direction register to enable some GPIO as output,
remaining are input */
p0.l = lo(PORTFIO_DIR);
p0.h = hi(PORTFIO_DIR);
r0.h = 0x0000;
r0.l = 0x0FC0;
w[p0] = r0;
ssync;
/* set port f clear register */
p0.l = lo(PORTFIO_CLEAR);
p0.h = hi(PORTFIO_CLEAR);
    r0.l = 0xFC0;
    w[p0] = r0;
    ssync;
/* set port f input enable register to enable input drivers of
some GPIOs */
p0.l = lo(PORTFIO_INEN);
p0.h = hi(PORTFIO_INEN);
r0.h = 0x0000;
r0.l = 0x003C;
w[p0] = r0;
ssync;

/* set port f polarity register */
p0.l = lo(PORTFIO_POLAR);
p0.h = hi(PORTFIO_POLAR);
r0 = 0x00000;
w[p0] = r0;
ssync;
```

8 GENERAL-PURPOSE TIMERS

This chapter describes the general-purpose (GP) timer module. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of GP timers for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For GP Timer interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

To determine how each of the GP Timers is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each GP Timer, refer to Chapter A, "System MMR Assignments".

GP timer behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 8-56

Overview

The general-purpose timers support the following operating modes:

- Single-shot mode for interval timing and single pulse generation
- Pulse width modulation (PWM) generation with consistent update of period and pulse width values
- External signal capture mode with consistent update of period and pulse width values
- External event counter mode

Feature highlights are:

- Synchronous operation
- Consistent management of period and pulse width values
- Interaction with PPI module for video frame sync operation
- Autobaud detection for UART module
- Graceful bit pattern termination when stopping
- Support for center-aligned PWM patterns
- Error detection on implausible pattern values
- All read and write accesses to 32-bit registers are atomic
- Every timer has its dedicated interrupt request output
- Unused timers can function as edge-sensitive pin interrupts

The internal structure of the individual timers is illustrated by Figure 8-1, which shows the details of timer 0 as a representative example. The other timers have identical structure.

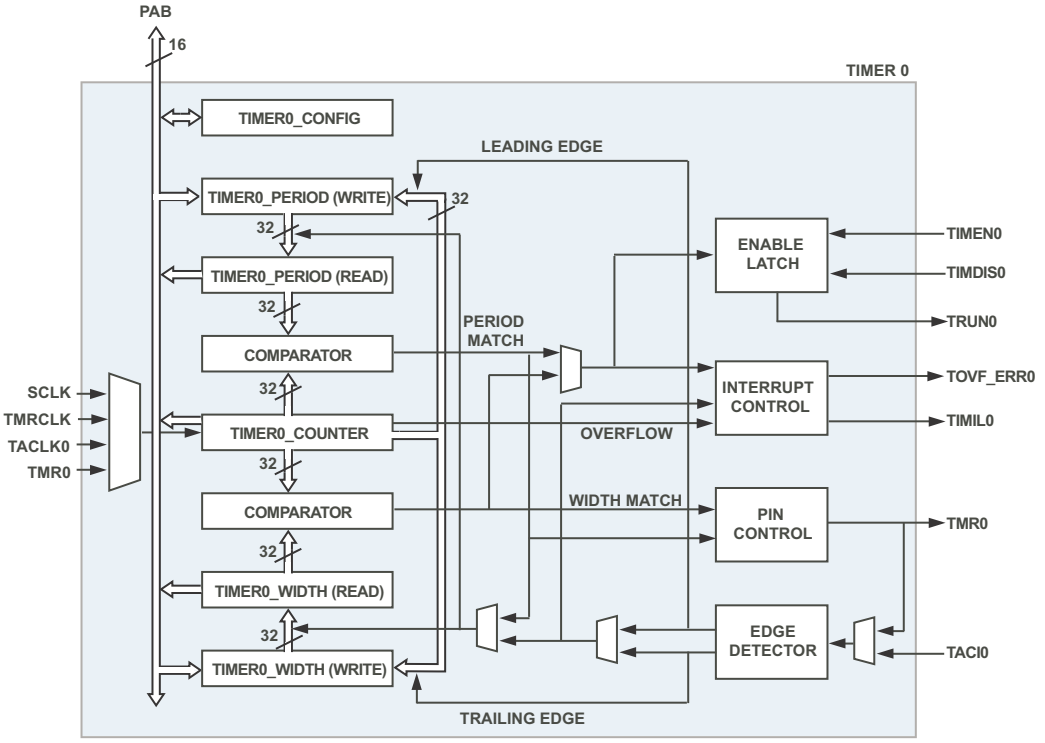


Figure 8-1. Internal Timer Structure

External Interface

Every timer has a dedicated TMR pin. If enabled, the TMR pins output the single-pulse or PWM signals generated by the timer. The TMR pins function as input in capture and counter modes. Polarity of the signals is programmable.

When clocked internally, the clock source is the processor's peripheral clock (SCLK). Assuming the peripheral clock is running at 100 MHz, the maximum period for the timer count is $((2^{32}-1) / 100\ \text{MHz}) = 42.9$ seconds.

Clock and capture input pins are sampled every SCLK cycle. The duration of every low or high state must be at least one SCLK. Therefore, the maximum allowed frequency of timer input signals is SCLK/2.

Internal Interface

Timer registers are always accessed by the core through the 16-bit PAB bus. Hardware ensures that all read and write operations from and to 32-bit timer registers are atomic.

Every timer has a dedicated interrupt request output that connects to the system interrupt controller (SIC).

Description of Operation

The core of every timer is a 32-bit counter, that can be interrogated through the read-only TIMER_COUNTER register. Depending on the mode of operation, the counter is reset to either 0x0000 0000 or 0x0000 0001 when the timer is enabled. The counter always counts upward. Usually, it is clocked by SCLK. In PWM mode it can be clocked by the alternate clock input TACLK or, alternatively, the common timer clock input TMRCLK. In counter mode, the counter is clocked by edges on the TMR input pin. The significant edge is programmable.

After 2^{32}-1 clocks, the counter overflows. This is reported by the overflow/error bit TOVF_ERR in the TIMER_STATUS register. In PWM and counter mode, the counter is reset by hardware when its content reaches the values stored in the TIMER_PERIOD register. In capture mode, the counter is reset by leading edges on the TMR or TACI input pin. If enabled, these events cause the interrupt latch TIMIL in the TIMER_STATUS register to be set and issue a system interrupt request. The TOVF_ERR and TIMIL latches are sticky and should be cleared by software using W1C (write-1-to-clear) operations to clear the interrupt request. The global TIMER_STATUS register

is 32-bits wide. A single atomic 32-bit read can report the status of all corresponding timers.

Before a timer can be enabled, its mode of operation is programmed in the individual timer-specific TIMER_CONFIG register. Then, the timers are started by writing a "1" to the representative bits in the global TIMER_ENABLE register.

The TIMER_ENABLE register can be used to enable all timers simultaneously. The register contains W1S (write-1-to-set) control bits, one for each timer. Correspondingly, the TIMER_DISABLE register contains W1C control bits to allow simultaneous or independent disabling of the timers. Either register can be read to check the enable status of the timers. A "1" indicates that the corresponding timer is enabled. The timer starts counting three SCLK cycles after the TIMEN bit is set.

While the PWM mode is used to generate PWM patterns, the capture mode (WDTH_CAP) is designed to “receive” PWM signals. A PWM pattern is represented by a pulse width and a signal period. This is described by the TIMER_WIDTH and TIMER_PERIOD register pair. In capture mode these registers are read only. Hardware always captures both values. Regardless of whether in PWM or capture mode, shadow buffers always ensure consistency between the TIMER_WIDTH and TIMER_PERIOD values. In PWM mode, hardware performs a plausibility check by the time the timer is enabled. If there is an error, the type is reported by the TIMER_CONFIG register and signalled by the TOVF_ERR bit.

Interrupt Processing

Each timer can generate a single interrupt. The resulting interrupt signals are routed to the system interrupt controller block for prioritization and masking. The timer status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source.

Figure 8-2 shows the interrupt structure of the timers.

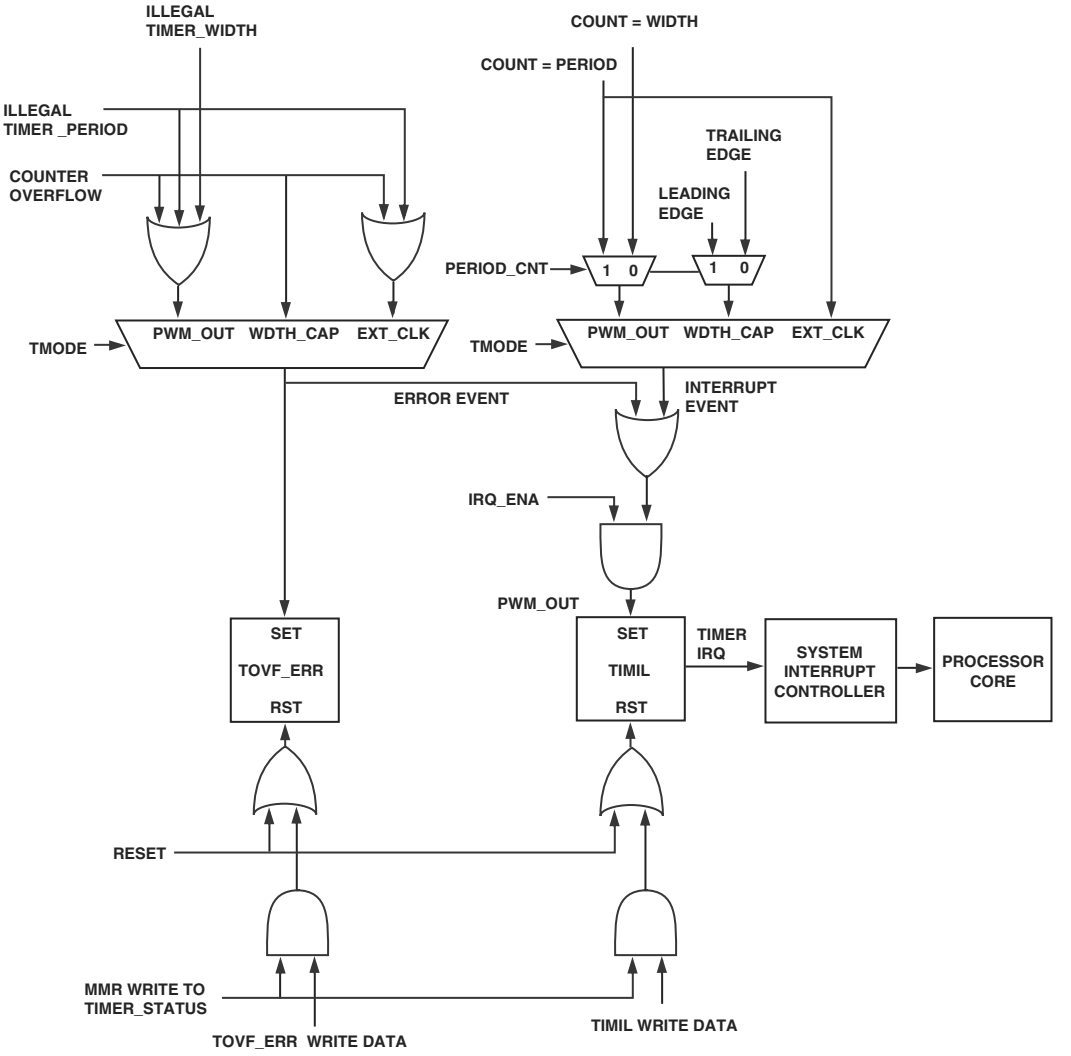


Figure 8-2. Timers Interrupt Structure

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the IMASK and SIC_IMASK registers. To poll the TIMIL bit without interrupt generation, set IRQ_ENA but leave the interrupt masked

at the system level. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions as reported by the TOVF_ERR bits.

The system interrupt controller enables flexible interrupt handling. All timers may or may not share the same CEC interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, multiple timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMIL latch bits at once by writing 0x000F 000F to the TIMER_STATUS register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMIL bit in the TIMER_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMIL clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the TIMIL bit in the TIMER_STATUS register at the very beginning of the interrupt service routine to avoid missing any timer events.

Illegal States

Every timer features an error detection circuit. It handles overflow situations but also performs pulse width vs. period plausibility checks. Errors are reported by the TOVF_ERR bits in the TIMER_STATUS register and the ERR_TYP bit field in the individual TIMER_CONFIG registers. Table 8-1 provides a summary of error conditions, using these terms:

- **Startup**. The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.
- **Rollover**. The time when the current count matches the value in TIMER_PERIOD and the counter is reloaded with the value "1".

- **Overflow**. The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged**. No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there has been no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads "0" if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads "1".

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write "1" to clear TOVF_ERR to acknowledge the error.

The following table can be read as: “In mode __ at event __, if TIMER_PERIOD is __ and TIMER_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __.”

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMR pin.

Table 8-1. Overview of Illegal States

Mode	Event	TIMER_PERIOD	TIMER_WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_CNT = 1	Startup (No boundary condition tests performed on TIMER_WIDTH)	== 0	Anything	b#10	Set
		== 1	Anything	b#10	Set
		≥ 2	Anything	No change	No change
	Rollover	== 0	Anything	b#10	Set
		== 1	Anything	b#11	Set
		≥ 2	== 0	b#11	Set
		≥ 2	< TIMER_PERIOD	No change	No change
		≥ 2	≥ TIMER_PERIOD	b#11	Set
	Overflow, not possible unless there is also another error, such as TIMER_PERIOD == 0	Anything	Anything	b#01	Set
PWM_OUT, PERIOD_CNT = 0	Startup	Anything	== 0	b#01	Set
		This case is not detected at startup, but results in an overflow error once the counter counts through its entire range.			
		Anything	≥ 1	No change	No change
	Rollover	Rollover is not possible in this mode.			
	Overflow, not possible unless there is also another error, such as TIMER_WIDTH == 0	Anything	Anything	b#01	Set

Table 8-1. Overview of Illegal States (Continued)

Mode	Event	TIMER_PERIOD	TIMER_WIDTH	ERR_TYP	TOVF_ERR
WDTH_CAP	Startup	TIMER_PERIOD and TIMER_WIDTH are read-only in this mode, no error possible.			
	Rollover	TIMER_PERIOD and TIMER_WIDTH are read-only in this mode, no error possible.			
	Overflow	Anything	Anything	b#01	Set
EXT_CLK	Startup	== 0	Anything	b#10	Set
		≥ 1	Anything	No change	No change
	Rollover	== 0	Anything	b#10	Set
		≥ 1	Anything	No change	No change
	Overflow, not possible unless there is also another error, such as TIMER_PERIOD == 0	Anything	Anything	b#01	Set

Modes of Operation

The following sections provide a functional description of the general-purpose timers in various operating modes.

Pulse Width Modulation (PWM_OUT) Mode

Use the PWM_OUT mode for PWM signal or single-pulse generation, for interval timing or for periodic interrupt generation. Figure 8-3 illustrates PWM_OUT mode.

Setting the TMODE field to b#01 in the TIMER_CONFIG register enables PWM_OUT mode. Here, the TMR pin is an output, but it can be disabled by setting the OUT_DIS bit in the TIMER_CONFIG register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

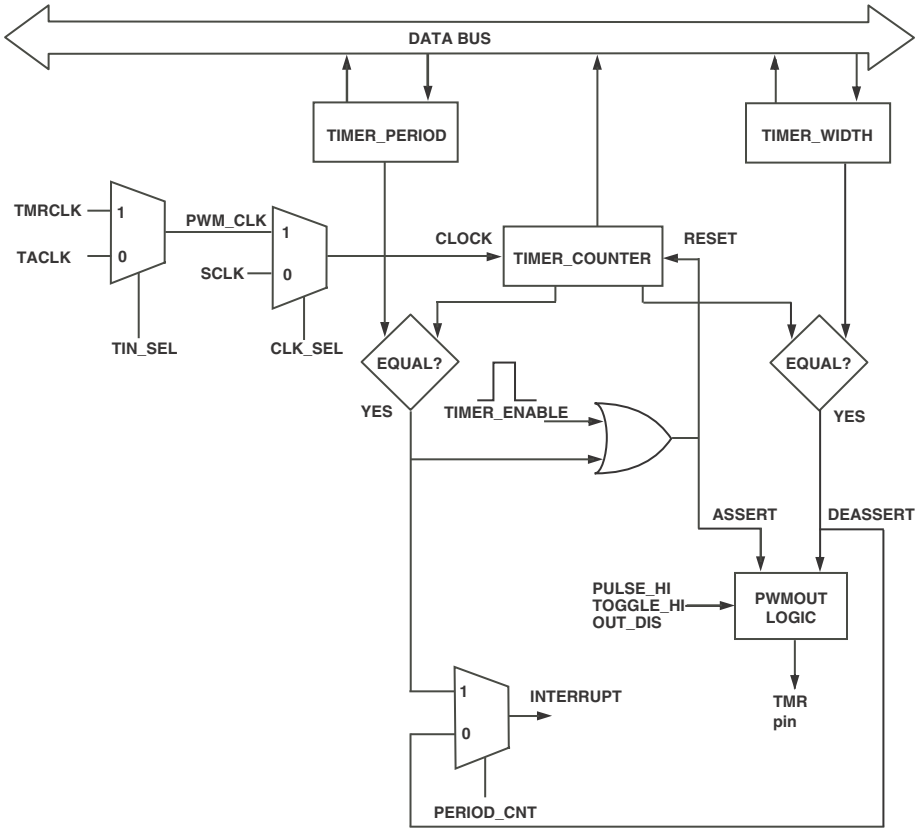


Figure 8-3. Timer Flow Diagram, PWM_OUT Mode

Once a timer has been enabled, the timer counter register is loaded with a starting value. If CLK_SEL = 0, the timer counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the timer period register. For either setting of CLK_SEL, when the timer counter equals the timer period, the timer counter is reset to 0x1 on the next clock.

In PWM_OUT mode, the PERIOD_CNT bit controls whether the timer generates one pulse or many pulses. When PERIOD_CNT is cleared (PWM_OUT single pulse mode), the timer uses the TIMER_WIDTH register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When PERIOD_CNT is set (PWM_OUT continuous pulse mode), the timer uses both the TIMER_PERIOD and TIMER_WIDTH registers and generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of PERIOD_CNT = 0 counts to the end of the width; a setting of PERIOD_CNT = 1 counts to the end of the period.

The TIMER_PERIOD and TIMER_WIDTH registers are read-only in some operation modes. Be sure to set the TMODE field in the TIMER_CONFIG register to b#01 before writing to these registers.

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the TIMER_CONFIG register. The TMR pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used. The TMR pin can also be disabled by the function enable and the multiplexer control registers.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMR pin. This mode can also be used to implement a precise delay.

The pulse width is defined by the TIMER_WIDTH register, and the TIMER_PERIOD register is not used. See Figure 8-4.

At the end of the pulse, the timer interrupt latch bit TIMIL is set, and the timer is stopped automatically. No writes to the TIMER_DISABLE register are required in this mode. If the PULSE_HI bit is set, an active high pulse is generated on the TMR pin. If PULSE_HI is not set, the pulse is active low.

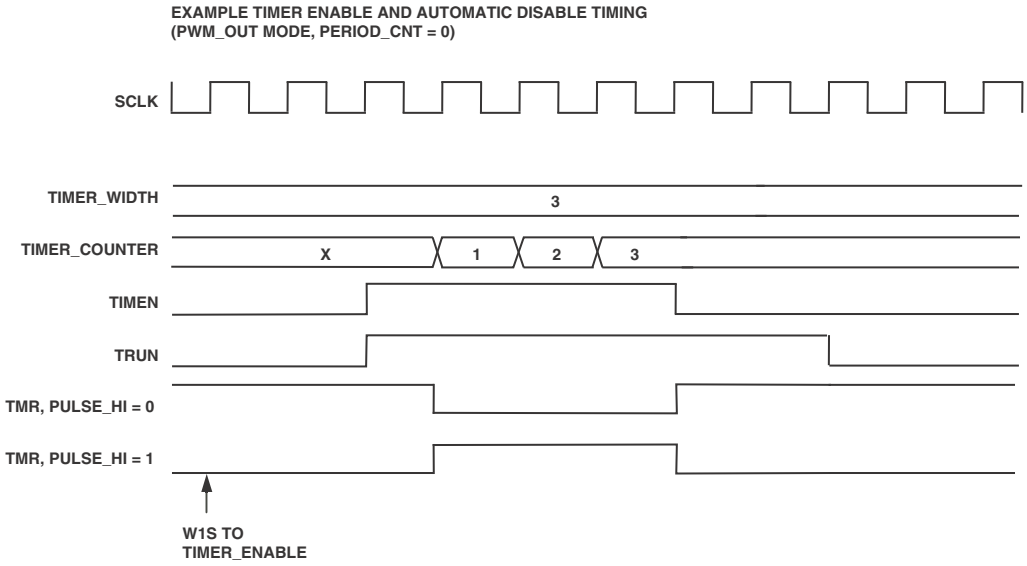


Figure 8-4. Timer Enable and Automatic Disable Timing

The pulse width may be programmed to any value from 1 to (2^{32}-1), inclusive.

Pulse Width Modulation Waveform Generation

If the PERIOD_CNT bit is set, the internally clocked timer generates rectangular signals with well-defined period and duty cycle (PWM patterns). This mode also generates periodic interrupts for real-time signal processing.

The 32-bit TIMER_PERIOD and TIMER_WIDTH registers are programmed with the values required by the PWM signal.

When the timer is enabled in this mode, the TMR pin is pulled to a deasserted state each time the counter equals the value of the pulse width register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the TMR pin, the PULSE_HI bit in the corresponding TIMER_CONFIG register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in PWM_OUT mode, the TMR pin is driven to the deasserted level.

Figure 8-5 shows timing details.

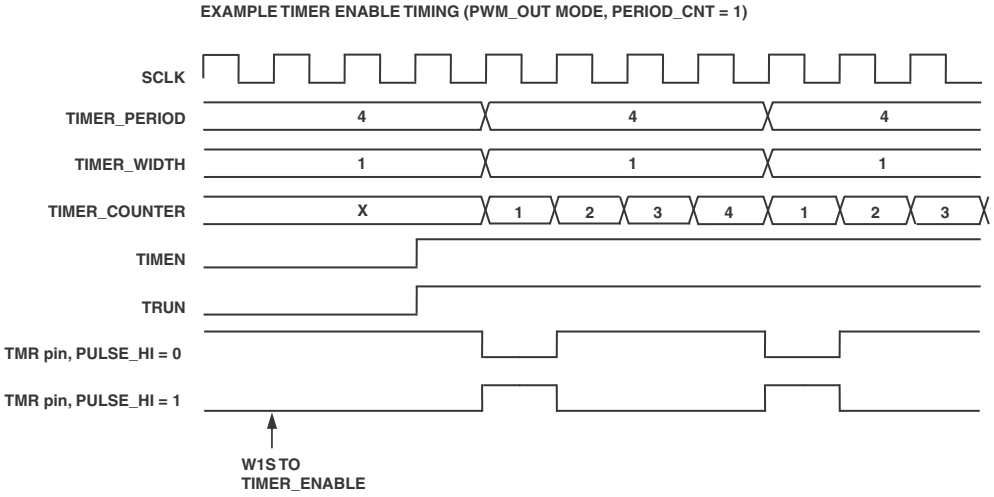


Figure 8-5. Timer Enable Timing

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine must clear the interrupt latch bit (TIMIL) and might alter period and/or width values. In PWM applications, the software needs to update period and pulse width values while the timer is

running. When software updates either the TIMER_PERIOD or TIMER_WIDTH registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. Reads from TIMER_PERIOD and TIMER_WIDTH registers return the old values until the period expires.

The TOVF_ERR status bit signifies an error condition in PWM_OUT mode. The TOVF_ERR bit is set if TIMER_PERIOD = 0 or TIMER_PERIOD = 1 at startup, or when the timer counter register rolls over. It is also set if the timer pulse width register is greater than or equal to the timer period register by the time the counter rolls over. The ERR_TYP bits are set when the TOVF_ERR bit is set.

Although the hardware reports an error if the TIMER_WIDTH value equals the TIMER_PERIOD value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore the TOVL_ERR flags. Pulse width values greater than the period value are not recommended. Similarly, TIMER_WIDTH = 0 is not a valid operation. Duty cycles of 0% are not supported.

To generate the maximum frequency on the TMR output pin, set the period value to “2” and the pulse width to "1". This makes the pin toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to (2^{32} – 1), inclusive. The pulse width may be programmed to any value from 1 to (period – 1), inclusive.

PULSE_HI Toggle Mode

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (via the TIMER_WIDTH register). When two or more timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 8-6.

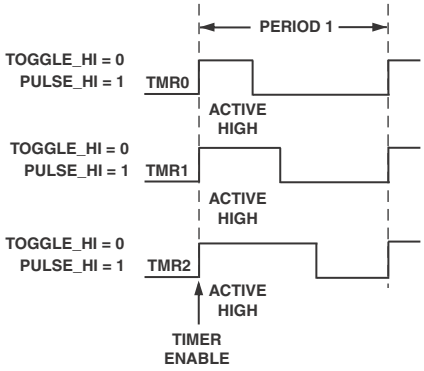


Figure 8-6. Example of Timers With Pulses Aligned to Asserting Edge

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a symmetrical rectangular waveform. The effective waveform is active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of the TOGGLE_HI bit has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.

In TOGGLE_HI mode, when PULSE_HI is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When PULSE_HI is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions when Count = Pulse Width. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

Figure 8-7 shows an example with three timers running with the same period settings. When software does not alter the PWM settings at run-time, the duty cycle is 50%. The values of the TIMER_WIDTH registers control the phase between the signals.

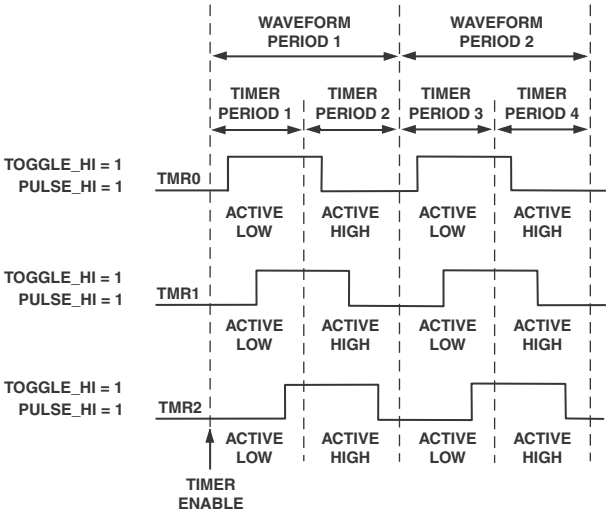


Figure 8-7. Three Timers With Same Period Settings

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (see Figure 8-8).

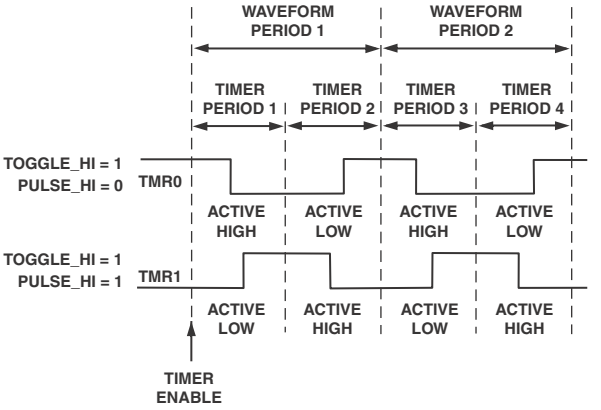


Figure 8-8. Two Timers With Non-overlapping Clocks

When TOGGLE_HI = 0, software updates the TIMER_PERIOD and TIMER_WIDTH registers once per waveform period. When TOGGLE_HI = 1, software updates the TIMER_PERIOD and TIMER_WIDTH registers twice per waveform. Period values are half as large. In odd-numbered periods, write (Period – Width) instead of Width to the TIMER_WIDTH register in order to obtain center-aligned pulses.

For example, if the pseudo-code when TOGGLE_HI = 0 is:

```
int period, width;
for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, period) ;
     write (TIMER_WIDTH, width) ;
}
```

Then when TOGGLE_HI = 1, the pseudo-code would be:

```
int period, width ;
int per1, per2, wid1, wid2 ;

for (;;) {
     period = generate_period(...) ;
     width = generate_width(...) ;

     per1 = period/2 ;
     wid1 = width/2 ;

     per2 = period/2 ;
     wid2 = width/2 ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, per1) ;
     write (TIMER_WIDTH, per1 - wid1) ;

     waitfor (interrupt) ;

     write (TIMER_PERIOD, per2) ;
     write (TIMER_WIDTH, wid2) ;

}
```

As shown in this example, the pulses produced do not need to be symmetric (wid1 does not need to equal wid2). The period can be offset to adjust the phase of the pulses produced (per1 does not need to equal per2).

The TRUN bit in the TIMER_STATUS register is updated only at the end of even-numbered periods in TOGGLE_HI mode. When TIMER_DISABLE is written to "1", the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when TOGGLE_HI = 0, errors are reported if the TIMER_PERIOD register is either set to “0” or "1", or when the width value is greater than or equal to the period value.

Externally Clocked PWM_OUT

By default, the timer is clocked internally by SCLK. Alternatively, if the CLK_SEL bit in the TIMER_CONFIG register is set, the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the TACLK pin, but may be taken from the common TMRCLK pin regardless of whether the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse width modulation waveforms or generates a single pulse with pulse width defined by the TIMER_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMR pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced. The timer interrupt may occur slightly before the corresponding edge on the TMR pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The alternate timer clock inputs (TACLK) are enabled when a timer is in PWM_OUT mode with CLK_SEL = 1 and TIN_SEL = 0, without regard to the content of the multiplexer control and function enable registers.

Using PWM_OUT Mode With the PPI

Some timers may be used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to "Frame Synchronization in GP Modes" on page 15-19.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLE) as a "stop is pending" condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMR pin. The processor can determine when the timer stops running by polling for the corresponding TRUN bit in the TIMER_STATUS register to read "0" or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (TIMER_CONFIG cannot be written to a new value) until after the timer stops and TRUN reads "0".

In PWM_OUT single pulse mode (PERIOD_CNT = 0), it is not necessary to write TIMER_DISABLE to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLE (and TIMER_DISABLE) is cleared, and the corresponding TRUN bit is cleared. See Figure 8-4 on page 8-13. To generate multiple pulses, write a "1" to TIMER_ENABLE, wait for the timer to stop, then write another "1" to TIMER_ENABLE.

In continuous PWM generation mode (PWM_OUT, PERIOD_CNT = 1) software can stop the timer by writing to the TIMER_DISABLE register. To prevent the ongoing PWM pattern from being stopped in an unpredictable way, the timer does not stop immediately when the corresponding "1" has been written to the TIMER_DISABLE register. Rather, the write simply clears the enable latch and the timer still completes the ongoing PWM patterns gracefully. It stops cleanly at the end of the first period when the enable latch is cleared. During this final period the TIMEN bit returns "0", but the TRUN bit still reads as a "1".

If the TRUN bit is not cleared explicitly, and the enable latch can be cleared and re-enabled all before the end of the current period will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself.

Figure 8-9 shows detailed timing.

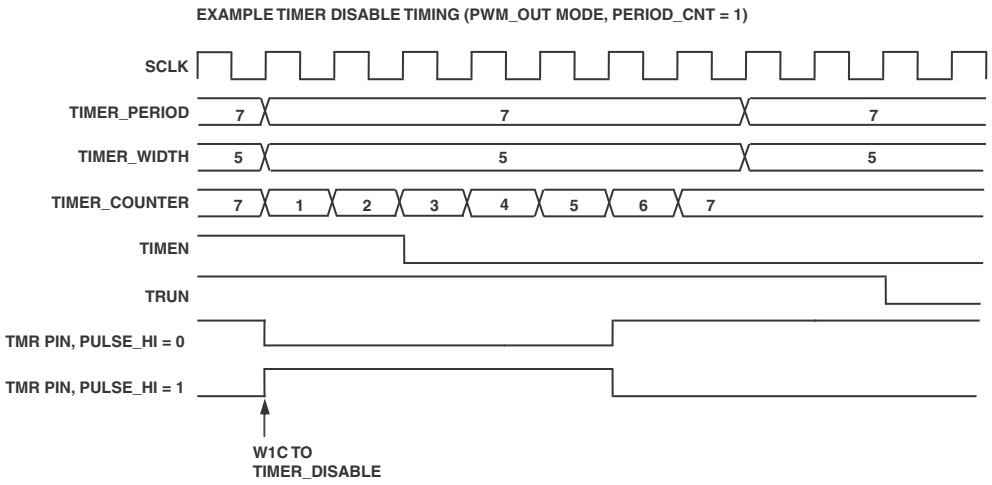


Figure 8-9. Timer Disable Timing

If necessary, the processor can force a timer in PWM_OUT mode to abort immediately. Do this by first writing a "1" to the corresponding bit in TIMER_DISABLE, and then writing a "1" to the corresponding TRUN bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMR pin.

When a timer is disabled, the TIMER_COUNTER register retains its state; when a timer is re-enabled, the timer counter is reinitialized based on the

operating mode. The TIMER_COUNTER register is read-only. Software cannot overwrite or preset the timer counter value directly.

Pulse Width Count and Capture (WDTH_CAP) Mode

Use the WDTH_CAP mode, often simply called “capture mode,” to measure pulse widths on the TMR or TACI input pins, or to “receive” PWM signals. Figure 8-10 shows a flow diagram for WDTH_CAP mode.

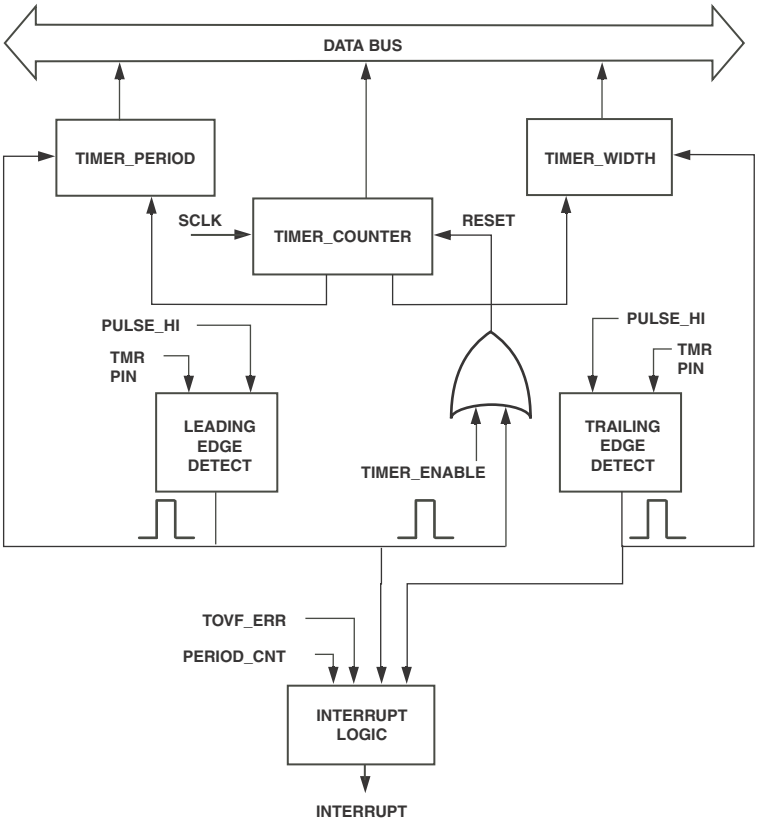


Figure 8-10. Timer Flow Diagram, WDTH_CAP Mode

In WDTH_CAP mode, the TMR pin is an input pin. The internally clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the TMODE field to b#10 in the TIMER_CONFIG register enables this mode.

When enabled in this mode, the timer resets the count in the TIMER_COUNTER register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMR pin.

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the TIMER_COUNTER register into the width buffer. At the next leading edge, the timer transfers the current 32-bit value of the TIMER_COUNTER register into the period buffer. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trailing edge of the TMR pin, the PULSE_HI bit in the TIMER_CONFIG register is set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the content of the counter register is captured to the pulse width buffer on the rising edge, and to the period buffer on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the counter value is captured to the pulse width buffer on the falling edge, and to the period buffer on the next rising edge.

In WDTH_CAP mode, these three events always occur at the same time:

1. The TIMER_PERIOD register is updated from the period buffer.
2. The TIMER_WIDTH register is updated from the width buffer.
3. The TIMIL bit gets set (if enabled) but does not generate an error.

The PERIOD_CNT bit in the TIMER_CONFIG register controls the point in time at which this set of transactions is executed. Taken together, these

three events are called a measurement report. The TOVF_ERR bit does not get set at a measurement report. A measurement report occurs, at most, once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMER_PERIOD and TIMER_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (see Figure 8-11), then the TIMER_PERIOD and TIMER_WIDTH registers report the pulse period and pulse width measured in the period that just ended. If the PERIOD_CNT bit is cleared and a trailing edge occurred (see Figure 8-12), then the TIMER_WIDTH register reports the pulse width measured in the pulse that just ended, but the TIMER_PERIOD register reports the pulse period measured at the end of the previous period.

If the PERIOD_CNT bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the TIMER_PERIOD value in this case returns "0", as shown in Figure 8-12. To measure the pulse width of a waveform that has only one leading edge and one trailing edge, set PERIOD_CNT = 0. If PERIOD_CNT = 1 for this case, no period value is captured in the period buffer. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps around. In this case, both TIMER_WIDTH and TIMER_PERIOD read "0"

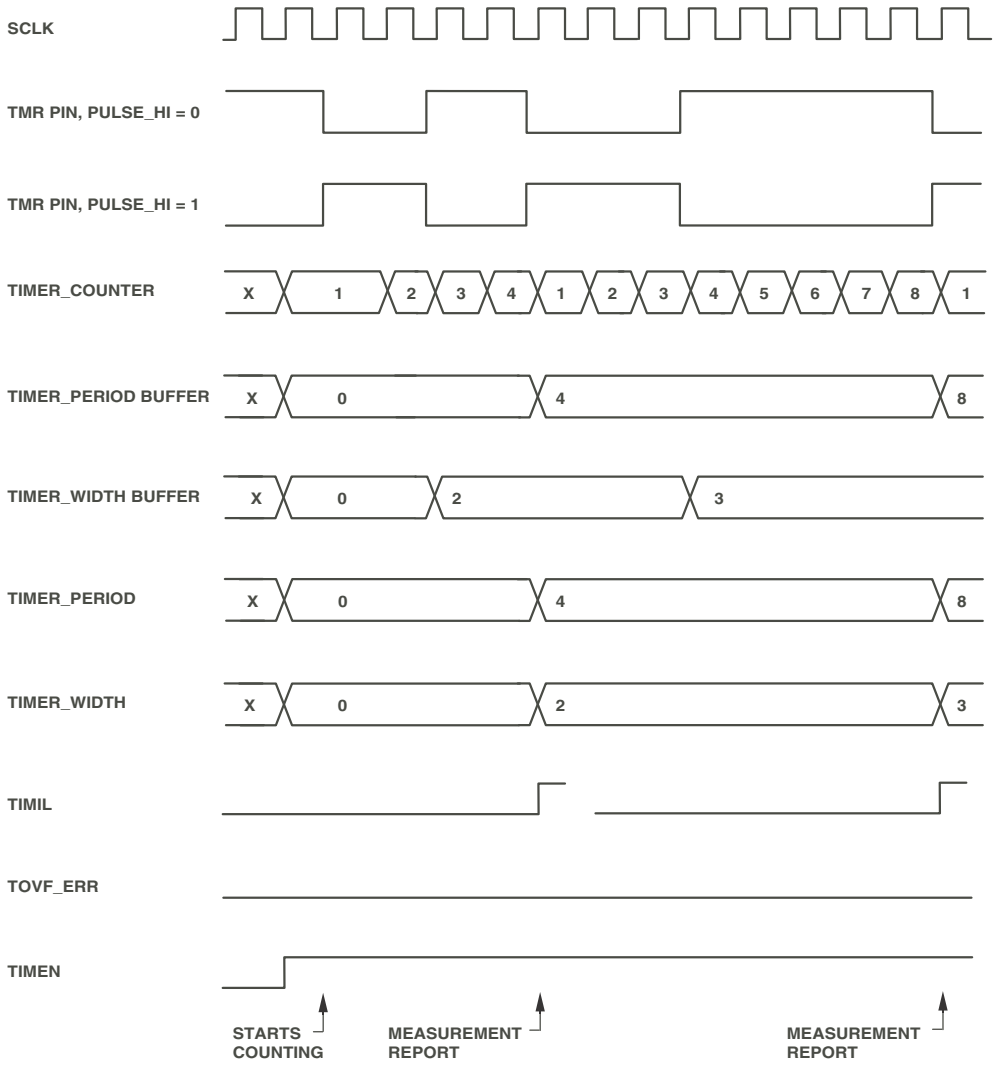


Figure 8-11. Example of Period Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 1)

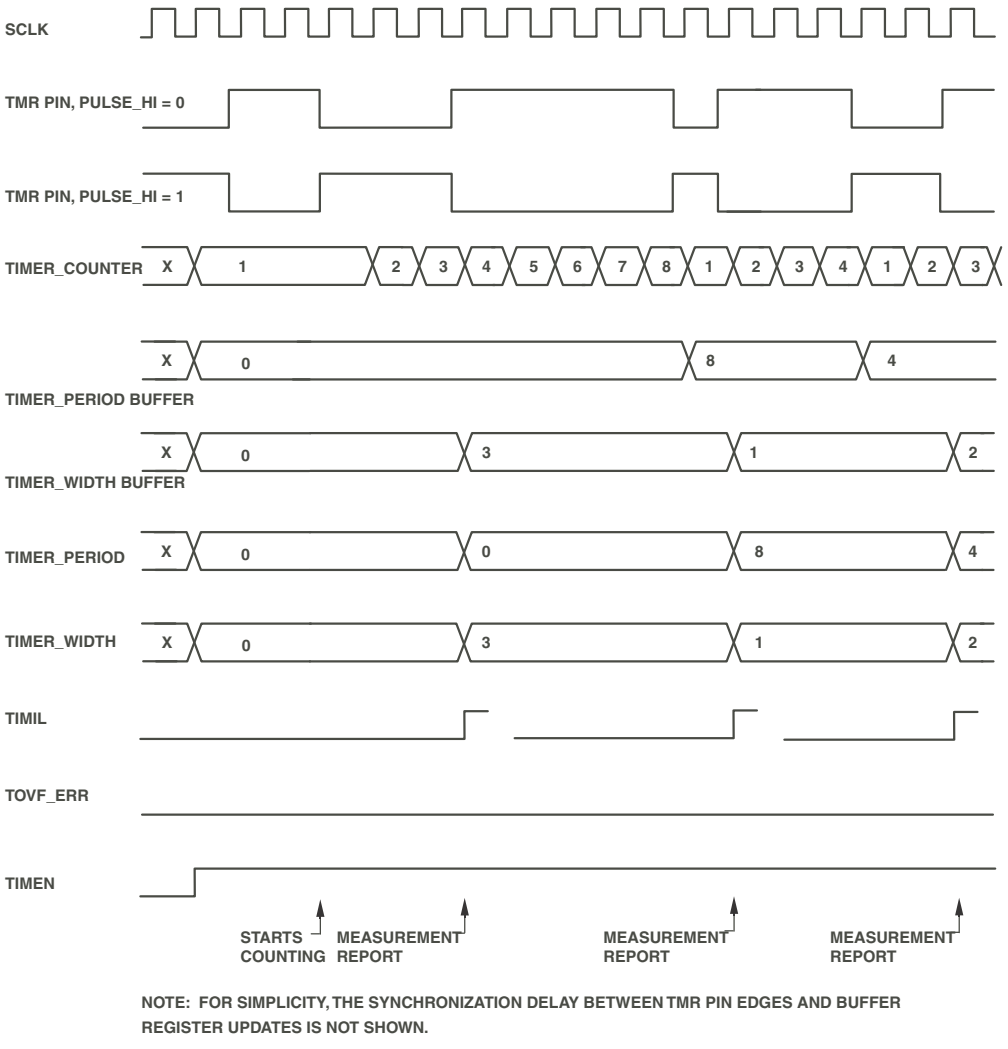


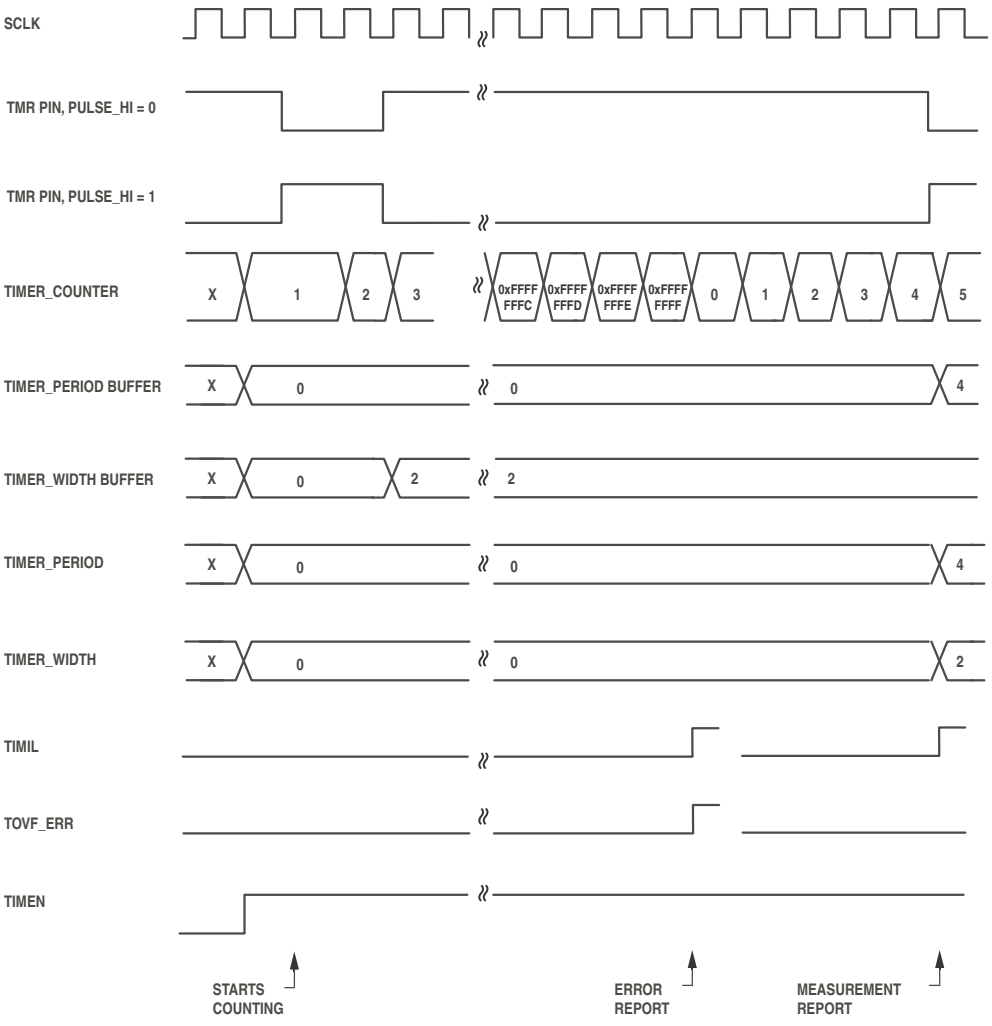
Figure 8-12. Example of Width Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 0)

(because no measurement report occurred to copy the value captured in the width buffer to TIMER_WIDTH). See the first interrupt in Figure 8-13.

When using the PERIOD_CNT = 0 mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be reenabled as appropriate in preparation for another measurement. This procedure prevents the timer from free-running after the width measurement, and from logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if the TIMER_COUNTER register wraps around from 0xFFFF FFFF to "0" in the absence of a leading edge. At that point, the TOVF_ERR bit in the TIMER_STATUS register and the ERR_TYP bits in the TIMER_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMER_PERIOD and TIMER_WIDTH registers are never updated at the time an error is signaled.

Refer to Figure 8-13 and Figure 8-14 for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMR PIN EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 8-13. Example Timing for Period Overflow Followed by Period Capture (`WDTH_CAP` mode, `PERIOD_CNT` = 1)

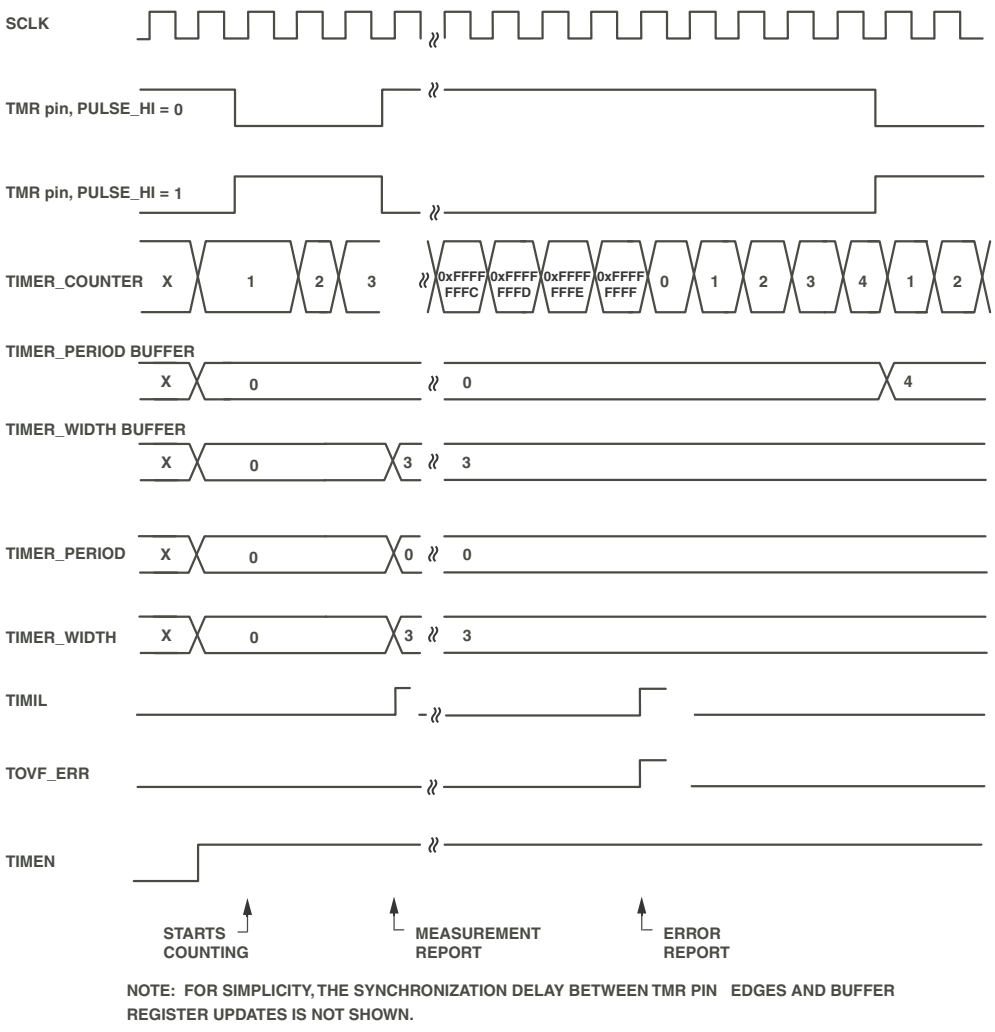


Figure 8-14. Example Timing for Width Capture Followed by Period Overflow (`WDTH_CAP` mode, `PERIOD_CNT` = 0)

Both TIMIL and TOVF_ERR are sticky bits, and software must explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMER_PERIOD nor the TIMER_WIDTH register were updated. If the timer overflowed and PERIOD_CNT = 0, the TIMER_PERIOD and TIMER_WIDTH registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full 2^{32} SCLK counts to the total for the period, but the width is ambiguous. For example, in Figure 8-13 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the TMR pin is not required to have a 50% duty cycle, but the minimum TMR pin low time is one SCLK period and the minimum TMR pin high time is one SCLK period. This implies the maximum TMR pin input frequency is SCLK/2 with a 50% duty cycle. Under these conditions, the WDTH_CAP mode timer would measure Period = 2 and Pulse Width = 1.

Autobaud Mode

On some devices, in WDTH_CAP mode, some of the timers can provide autobaud detection for the Universal Asynchronous Receiver/Transmitter (UART) interface(s). The TIN_SEL bit in the TIMER_CONFIG register causes the timer to sample the TACI pin instead of the TMR pin when enabled for WDTH_CAP mode. Autobaud detection can be used for initial bit rate negotiations as well as for detection of bit rate drifts while the interface is in operation.

External Event (EXT_CLK) Mode

Use the EXT_CLK mode (sometimes referred to as the counter mode) to count external events—that is, signal edges on the TMR pin (which is an input in this mode). Figure 8-15 shows a flow diagram for EXT_CLK mode.

The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in TIMER_COUNTER represents the number of leading edge events detected. Setting the TMODE field to b#11 in the TIMER_CONFIG register enables this mode. The TIMER_PERIOD register is programmed with the value of the maximum timer external count.

The waveform applied to the TMR pin is not required to have a 50% duty cycle, but the minimum TMR low time is one SCLK period, and the minimum TMR high time is one SCLK period. This implies the maximum TMR pin input frequency is SCLK/2.

Period may be programmed to any value from 1 to (2^{32} – 1), inclusive.

After the timer has been enabled, it resets the TIMER_COUNTER register to 0x0 and then waits for the first leading edge on the TMR pin. This edge causes the TIMER_COUNTER register to be incremented to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMIL bit is set, and an interrupt is generated. The next leading edge reloads the TIMER_COUNTER register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

The configuration bits TIN_SEL and PERIOD_CNT have no effect in this mode. The TOVF_ERR and ERR_TYP bits are set if the TIMER_COUNTER register wraps around from 0xFFFF FFFF to "0" or if Period = "0" at startup or when the TIMER_COUNTER register rolls over (from Count = Period to Count = 0x1). The TIMER_WIDTH register is unused.

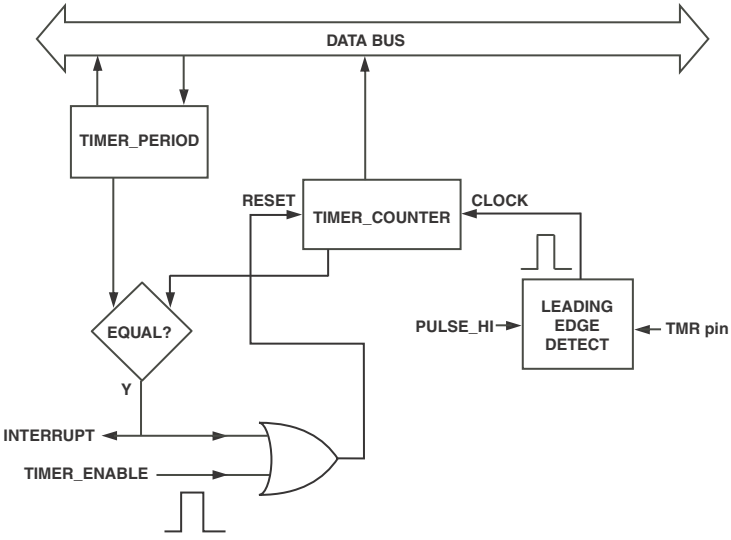


Figure 8-15. Timer Flow Diagram, EXT_CLK Mode

Programming Model

The architecture of the timer block enables any of the timers within this block to work individually or synchronously along with others as a group of timers. Regardless of the operating mode, the programming model is always straightforward. Because of the error checking mechanism, always follow this order when enabling timers:

1. Set timer mode.
2. Write `TIMER_WIDTH` and `TIMER_PERIOD` registers as applicable.
3. Enable timer.

If this order is not followed, the plausibility check may fail because of undefined width and period values, or writes to `TIMER_WIDTH` and `TIMER_PERIOD` may result in an error condition, because the registers are read-only in some modes. The timer may not start as expected.

If in PWM_OUT mode the PWM patterns of the second period differ from the patterns of the first one, the initialization sequence above might become:

1. Set timer mode to PWM_OUT.
2. Write first TIMER_WIDTH and TIMER_PERIOD value pair.
3. Enable timer.
4. Immediately write second TIMER_WIDTH and TIMER_PERIOD value pair.

Hardware ensures that the buffered width and period values become active when the first period expires.

Once started, timers require minimal interaction with software, which is usually performed by an interrupt service routine. In PWM_OUT mode software must update the pulse width and/or settings as required. In WDTH_CAP mode it must store captured values for further processing. In any case, the service routine should clear the TIMIL bits of the timers it controls.

Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of multiple identical timer units.

Each timer provides four registers:

- TIMER_CONFIG[15:0] – timer configuration register
- TIMER_WIDTH[31:0] – timer pulse width register
- TIMER_PERIOD[31:0] – timer period register
- TIMER_COUNTER[31:0] – timer counter register

Additionally, three registers are shared between the timers within a block:

- TIMER_ENABLE[15:0] – timer enable register
- TIMER_DISABLE[15:0] – timer disable register
- TIMER_STATUS[31:0] – timer status register

The size of accesses is enforced. A 32-bit access to a TIMER_CONFIG register or a 16-bit access to a TIMER_WIDTH, TIMER_PERIOD, or TIMER_COUNTER register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the TIMER_ENABLE, TIMER_DISABLE, and TIMER_STATUS registers. On a 32-bit read of one of the 16-bit registers, the upper word returns all 0s.

Timer Enable Register (TIMER_ENABLE)

Figure 8-16 shows an example of the TIMER_ENABLE register for a product with eight timers. The register allows simultaneous enabling of multiple timers so that they can run synchronously. For each timer there is a single W1S control bit. Writing a "1" enables the corresponding timer; writing a "0" has no effect. The bits can be set individually or in any combination. A read of the TIMER_ENABLE register shows the status of the enable for the corresponding timer. A "1" indicates that the timer is enabled. All unused bits return "0" when read.

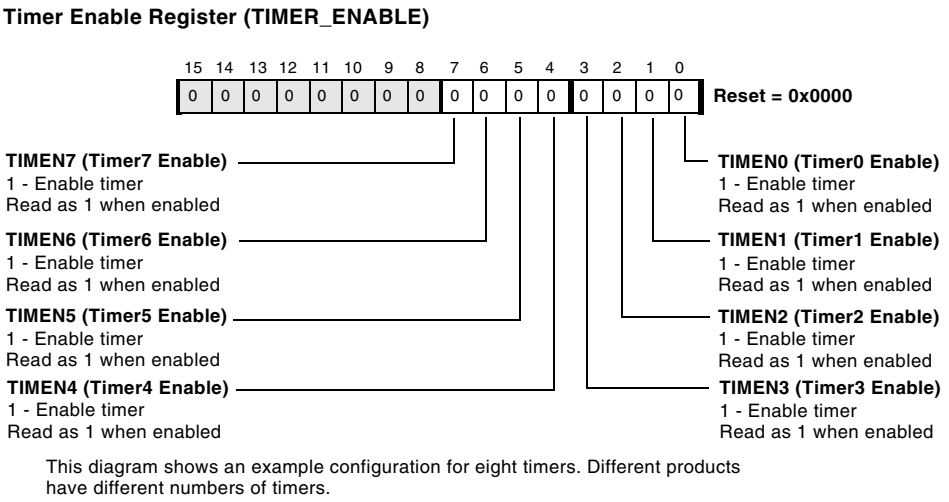
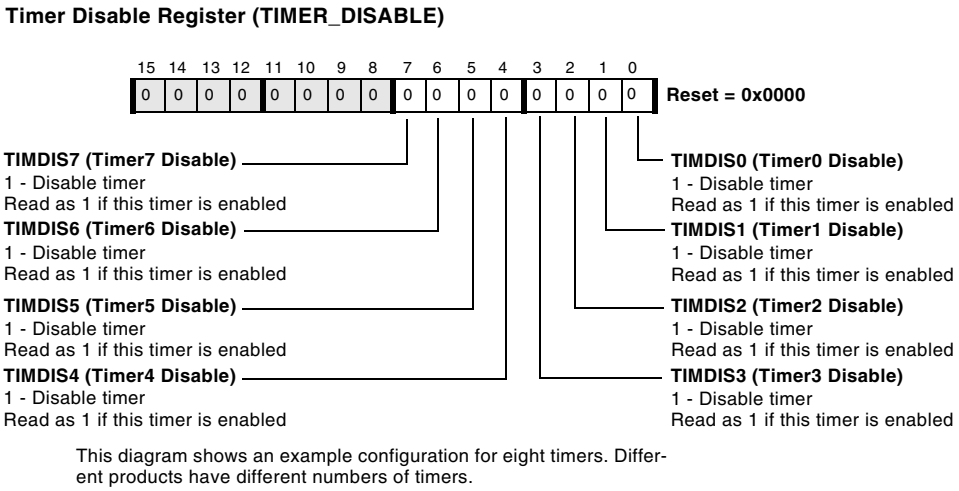

Timer Enable Register (TIMER_ENABLE)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TIMEN7 (Timer7 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN6 (Timer6 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN5 (Timer5 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN4 (Timer4 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN0 (Timer0 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN1 (Timer1 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN2 (Timer2 Enable)
1 - Enable timer
Read as 1 when enabled
TIMEN3 (Timer3 Enable)
1 - Enable timer
Read as 1 when enabled
This diagram shows an example configuration for eight timers. Different products have different numbers of timers.

Figure 8-16. Timer Enable Register

Timer Disable Register (TIMER_DISABLE)

Figure 8-17 shows an example of the TIMER_DISABLE register for a product with eight timers. The register allows simultaneous disabling of multiple timers. For each timer there is a single W1C control bit. Writing a "1" disables the corresponding timer; writing a "0" has no effect. The bits can be cleared individually or in any combination. A read of the TIMER_DISABLE register returns a value identical to a read of the TIMER_ENABLE register. A "1" indicates that the timer is enabled. All unused bits return "0" when read.


Timer Disable Register (TIMER_DISABLE)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TIMDIS7 (Timer7 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS6 (Timer6 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS5 (Timer5 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS4 (Timer4 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS0 (Timer0 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS1 (Timer1 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS2 (Timer2 Disable)
1 - Disable timer
Read as 1 if this timer is enabled
TIMDIS3 (Timer3 Disable)
1 - Disable timer
Read as 1 if this timer is enabled

This diagram shows an example configuration for eight timers. Different products have different numbers of timers.

Figure 8-17. Timer Disable Register

In PWM_OUT mode, a write of a "1" to TIMER_DISABLE does not stop the corresponding timer immediately. Rather, the timer continues running and stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse (if PERIOD_CNT = 0). If necessary, the processor can force a timer in PWM_OUT mode to stop immediately by first writing a "1" to the corresponding bit in TIMER_DISABLE, and then writing a "1" to the corresponding TRUN bit in TIMER_STATUS. See "Stopping the Timer in PWM_OUT Mode" on page 8-21.

In WDTH_CAP and EXT_CLK modes, a write of a "1" to TIMER_DISABLE stops the corresponding timer immediately.

Timer Status Register (TIMER_STATUS)

The TIMER_STATUS register indicates the status of the timers and is used to check the status of multiple timers with a single read. Status bits are sticky and W1C. The TRUN bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a TIMER_STATUS register read access, all reserved or unused bits return a "0". Figure 8-18 on page 8-39 shows an example of the TIMER_STATUS register for a product with eight timers.

For detailed behavior and usage of the TRUN bit see "Stopping the Timer in PWM_OUT Mode" on page 8-21. Writing the TRUN bits has no effect in other modes or when a timer has not been enabled. Writing the TRUN bits to "1" in PWM_OUT mode has no effect on a timer that has not first been disabled.

Error conditions are explained in "Illegal States" on page 8-7.

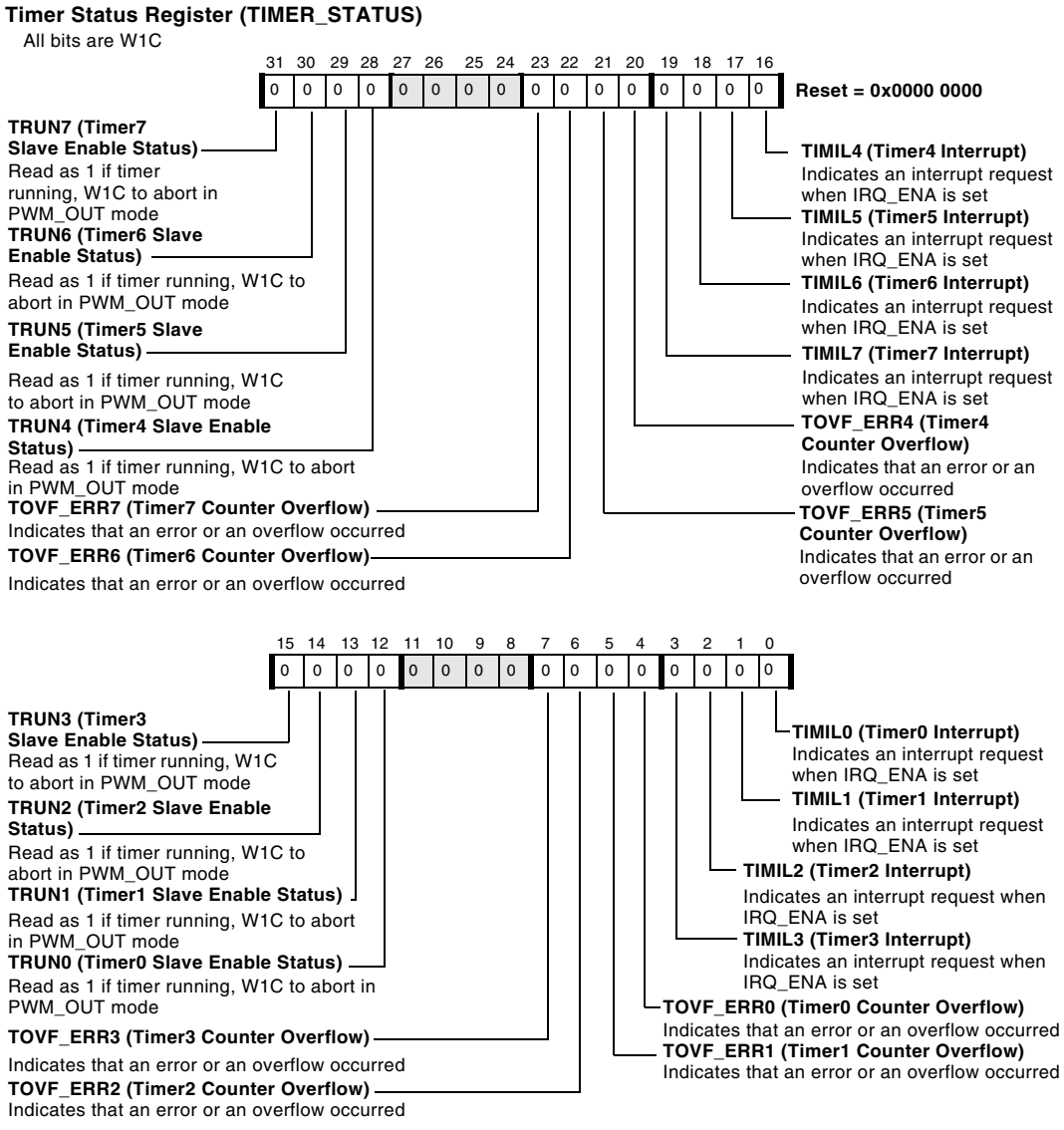

Timer Status Register (TIMER_STATUS)
All bits are W1C
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
TRUN7 (Timer7 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN6 (Timer6 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN5 (Timer5 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN4 (Timer4 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TOVF_ERR7 (Timer7 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR6 (Timer6 Counter Overflow)
Indicates that an error or an overflow occurred
TIMIL4 (Timer4 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL5 (Timer5 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL6 (Timer6 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL7 (Timer7 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TOVF_ERR4 (Timer4 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR5 (Timer5 Counter Overflow)
Indicates that an error or an overflow occurred
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
TRUN3 (Timer3 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN2 (Timer2 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN1 (Timer1 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TRUN0 (Timer0 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode
TOVF_ERR3 (Timer3 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR2 (Timer2 Counter Overflow)
Indicates that an error or an overflow occurred
TIMIL0 (Timer0 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL1 (Timer1 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL2 (Timer2 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TIMIL3 (Timer3 Interrupt)
Indicates an interrupt request when IRQ_ENA is set
TOVF_ERR0 (Timer0 Counter Overflow)
Indicates that an error or an overflow occurred
TOVF_ERR1 (Timer1 Counter Overflow)
Indicates that an error or an overflow occurred
This diagram shows an example configuration for eight timers. Different products have different numbers of timers, therefore some of the bits may not be applicable to your device.

Figure 8-18. Timer Status Register

Timer Configuration Register (TIMER_CONFIG)

The operating mode for each timer is specified by its TIMER_CONFIG register. The TIMER_CONFIG register, shown in Figure 8-19, may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUN bit in TIMER_STATUS before attempting to reprogram TIMER_CONFIG. The TIMER_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled.

Each time TOVF_ERR is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 8-1 on page 8-9. The TIMER_CONFIG register also controls the behavior of the TMR pin, which becomes an output in PWM_OUT mode (TMODE = 01) when the OUT_DIS bit is cleared.

When operating the PPI in GP output modes with internal frame syncs, the CLK_SEL and the TIN_SEL bits for the timers involved must be set to "1".

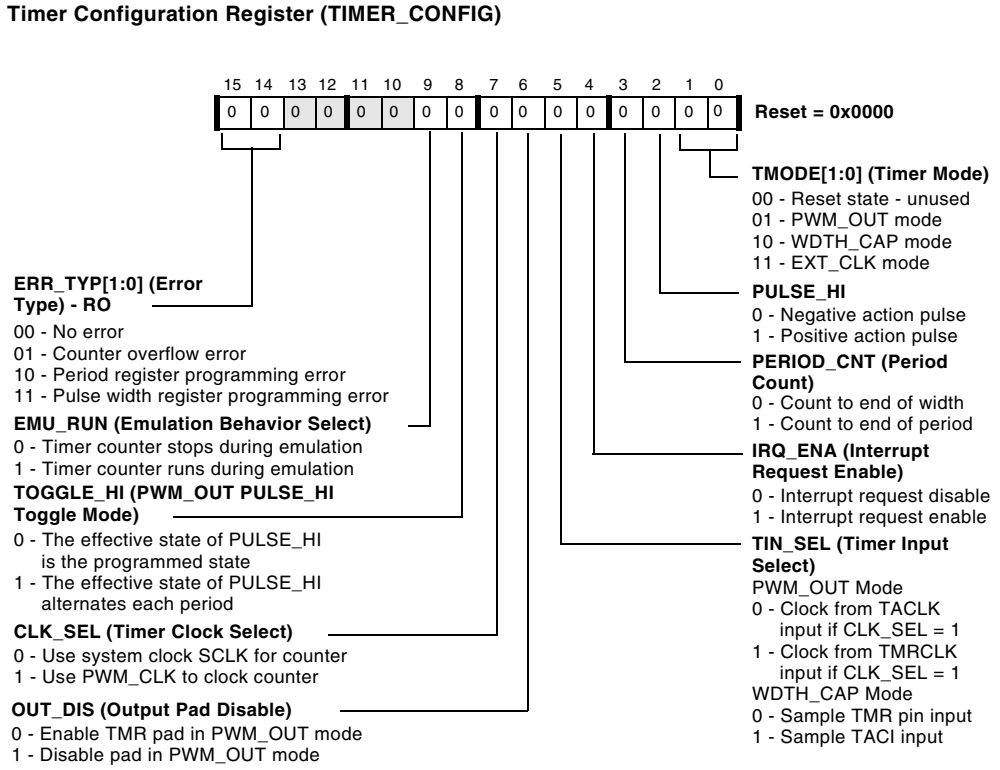

Timer Configuration Register (TIMER_CONFIG)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TMODE[1:0] (Timer Mode)
00 - Reset state - unused
01 - PWM_OUT mode
10 - WDTH_CAP mode
11 - EXT_CLK mode
PULSE_HI
0 - Negative action pulse
1 - Positive action pulse
PERIOD_CNT (Period Count)
0 - Count to end of width
1 - Count to end of period
IRQ_ENA (Interrupt Request Enable)
0 - Interrupt request disable
1 - Interrupt request enable
TIN_SEL (Timer Input Select)
PWM_OUT Mode
0 - Clock from TACLK input if CLK_SEL = 1
1 - Clock from TMRCLK input if CLK_SEL = 1
WDTH_CAP Mode
0 - Sample TMR pin input
1 - Sample TACI input
ERR_TYP[1:0] (Error Type) - RO
00 - No error
01 - Counter overflow error
10 - Period register programming error
11 - Pulse width register programming error
EMU_RUN (Emulation Behavior Select)
0 - Timer counter stops during emulation
1 - Timer counter runs during emulation
TOGGLE_HI (PWM_OUT PULSE_HI Toggle Mode)
0 - The effective state of PULSE_HI is the programmed state
1 - The effective state of PULSE_HI alternates each period
CLK_SEL (Timer Clock Select)
0 - Use system clock SCLK for counter
1 - Use PWM_CLK to clock counter
OUT_DIS (Output Pad Disable)
0 - Enable TMR pad in PWM_OUT mode
1 - Disable pad in PWM_OUT mode

Figure 8-19. Timer Configuration Register

Timer Counter Register (TIMER_COUNTER)

This read-only register retains its state when disabled. When enabled, the TIMER_COUNTER register is reinitialized by hardware based on configuration and mode. The TIMER_COUNTER register, shown in Figure 8-20, may be read at any time (whether the timer is running or stopped), and it returns an atomic 32-bit value. Depending on the operating mode, the incrementing counter can be clocked by four different sources: SCLK, the TMR pin, the alternative timer clock pin TACLK, or the common TMRCLK pin, which is most likely used as the PPI clock (PPI_CLK).

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMER_COUNTER register also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMR pin waveform is "stretched"; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on the TMR pin may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMER_PERIOD and TIMER_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMER_CONFIG to enable this behavior.

Timer Counter Register (TIMER_COUNTER)

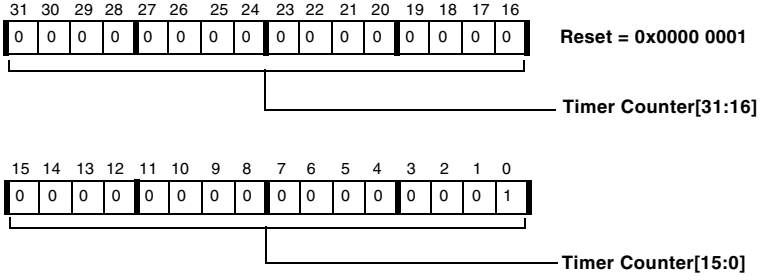


Figure 8-20. Timer Counter Register

Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers

When a timer is enabled and running, and the software writes new values to the TIMER_PERIOD register and the TIMER_WIDTH register, the writes are buffered and do not update the registers until the end of the current period (when TIMER_COUNTER equals TIMER_WIDTH).

Usage of the TIMER_PERIOD register, shown in Figure 8-21, and the TIMER_WIDTH register, shown in Figure 8-22, varies depending on the mode of the timer:

- In PWM_OUT mode, both the TIMER_PERIOD and TIMER_WIDTH register values can be updated “on-the-fly” since the values change simultaneously.
- In WDTH_CAP mode, the timer period and timer pulse width buffer values are captured at the appropriate time. The TIMER_PERIOD and TIMER_WIDTH registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.
- In EXT_CLK mode, the TIMER_PERIOD register is writable and can be updated “on-the-fly.” The TIMER_WIDTH register is not used.

If new values are not written to the TIMER_PERIOD register or the TIMER_WIDTH register, the value from the previous period is reused. Writes to the 32-bit TIMER_PERIOD register and TIMER_WIDTH register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the TIMER_PERIOD registers or TIMER_WIDTH registers are always stored in the buffer registers. Reads from the TIMER_PERIOD or TIMER_WIDTH registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the TIMER_PERIOD and TIMER_WIDTH registers are updated from their respective buffers at the end of the current period. See Figure 8-1 on page 8-3.

When the timer is disabled, writes to the buffer registers are immediately copied through to the TIMER_PERIOD or TIMER_WIDTH register so that they will be ready for use in the first timer period. For example, to change the values for the TIMER_PERIOD and/or TIMER_WIDTH registers in order to use a

different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In PWM_OUT mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the TIMER_PERIOD register and the TIMER_WIDTH register. The next period may use one old value and one new value. In order to prevent "pulse width ≥ period" errors, write the TIMER_WIDTH register before the TIMER_PERIOD register when decreasing the values, and write the TIMER_PERIOD register before the TIMER_WIDTH register when increasing the value.

Timer Period Register (TIMER_PERIOD)

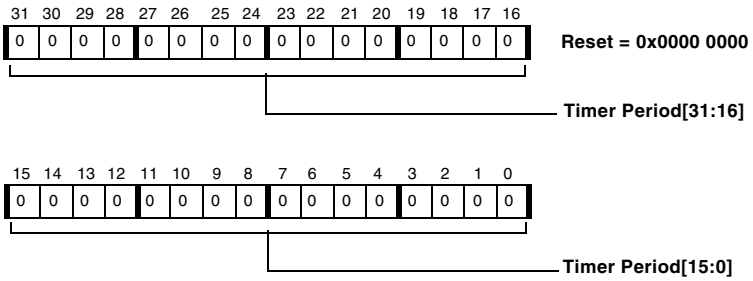


Figure 8-21. Timer Period Register

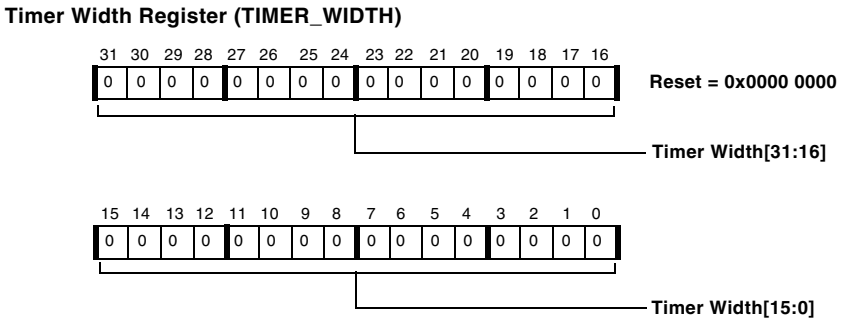


Figure 8-22. Timer Width Register

Summary

Table 8-2 summarizes control bit and register usage in each timer mode.

Table 8-2. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLE	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLE	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - Interrupt after measuring period 0 - Interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt

Table 8-2. Control Bit and Register Usage Chart (Continued)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIN_SEL	Depends on CLK_SEL: If CLK_SEL = 1, 1 - Count TMRCLK clocks 0 - Count TACLK clocks If CLK_SEL = 0, Unused	1 - Select TACI input 0 - Select TMR pin input	Unused
OUT_DIS	1 - Disable TMR pin 0 - Enable TMR pin	Unused	Unused
CLK_SEL	1 - PWM_CLK clocks timer 0 - SCLK clocks timer	Unused	Unused
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports b#00, b#01, b#10, or b#11, as appropriate	Reports b#00 or b#01, as appropriate	Reports b#00, b#01, or b#10, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused

Table 8-2. Control Bit and Register Usage Chart (Continued)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
Counter	RO: Counts up on SCLK or PWM_CLK	RO: Counts up on SCLK	RO: Counts up on TMR pin event
TRUN	Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect
TOVF_ERR	Set at startup or rollover if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or rollover if period = 0
IRQ	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Programming Examples

Listing 8-1 configures the port control registers in a way that enables `TMR` pins associated with Port G. This example assumes `TMR1-7` are connected to Port G bits 5–11.

Listing 8-1. Port Setup

```
timer_port_setup:
```

```
    [--sp] = (r7:7, p5:5);
    p5.h = hi(PORTG_FER);
    p5.l = lo(PORTG_FER);
    r7.l = PG5|PG6|PG7|PG8|PG9|PG10|PG11;
    w[p5] = r7;
    p5.l = lo(PORTG_MUX);
    r7.l = PFTE;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer_port_setup.end:
```

Listing 8-2 generates signals on the TMR4 and TMR5 outputs. By default, timer 5 generates a continuous PWM signal with a duty cycle of 50% (period = 0x40 SCLKs, width = 0x20 SCLKs) while the PWM signal generated by timer 4 has the same period but 25% duty cycle (width = 0x10 SCLKs).

If the preprocessor constant SINGLE_PULSE is defined, every TMR pin outputs only a single high pulse of 0x20 (timer 4) and 0x10 SCLKs (timer 5) duration.

In any case the timers are started synchronously and the rising edges are aligned. That is, the pulses are left aligned.

Listing 8-2. Signal Generation

```
// #define SINGLE_PULSE
timer45_signal_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = PULSE_HI | PWM_OUT;
#else
```

```
    r7.l = PERIOD_CNT | PULSE_HI | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x10 (z);
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7 = 0x20 (z);
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
#ifndef SINGLE_PULSE
    r7 = 0x40 (z);
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r7;
#endif
    r7.l = TIMEN5 | TIMEN4;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer45_signal_generation.end:
```

All subsequent examples use interrupts. Thus, Listing 8-3 illustrates how interrupts are generated and how interrupt service routines can be registered. In this example, the timer 5 interrupt is assigned to the IVG12 interrupt channel of the CEC controller.

Listing 8-3. Interrupt Setup

```
timer5_interrupt_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(IMASK);
    p5.l = lo(IMASK);
/* register interrupt service routine */
    r7.h = hi(isr_timer5);
    r7.l = lo(isr_timer5);
    [p5 + EVT12 - IMASK] = r7;
/* unmask IVG12 in CEC */
```

```
    r7 = [p5];
    bitset(r7, bitpos(EVT_IVG12));
    [p5] = r7;
/* assign timer 5 IRQ (= IRQ37 in this example) to IVG12 */
    p5.h = hi(SIC_IAR4);
    p5.l = lo(SIC_IAR4);
/*SIC_IAR register mapping is processor dependent*/
    r7.h = 0xFF5F;
    r7.l = 0xFFFF;
    [p5] = r7;
/* enable timer 5 IRQ */
    p5.h = hi(SIC_IMASK1);
    p5.l = lo(SIC_IMASK1);
/*SIC_IMASK register mapping is processor dependent*/
    r7 = [p5];
    bitset(r7, 5);
    [p5] = r7;
/* enable interrupt nesting */
    (r7:7, p5:5) = [sp++];
    [--sp] = reti;
    rts;
timer5_interrupt_setup.end:
```

The example shown in Listing 8-4 does not drive the TMR pin. It generates periodic interrupt requests every 0x1000 SCLK cycles. If the preprocessor constant SINGLE_PULSE was defined, timer 5 requests an interrupt only once. Unlike in a real application, the purpose of the interrupt service routine shown in this example is just the clearing of the interrupt request and counting interrupt occurrences.

Listing 8-4. Periodic Interrupt Requests

```
// #define SINGLE_PULSE
timer5_interrupt_generation:
    [--sp] = (r7:7, p5:5);
```

```
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = EMU_RUN | IRQ_ENA | OUT_DIS | PWM_OUT;
#else
    r7.l = EMU_RUN | IRQ_ENA | PERIOD_CNT | OUT_DIS | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x1000 (z);
#ifndef SINGLE_PULSE
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    r7 = 0x1 (z);
#endif
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7.l = TIMEN5;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    r0 = 0 (z);
    rts;
timer5_interrupt_generation.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r0+= 1;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 8-5 illustrates how two timers can generate two non-overlapping clock pulses as typically required for break-before-make scenarios. Both timers are running in `PWM_OUT` mode with `PERIOD_CNT` = 1 and `PULSE_HI` = 1.

Figure 8-23 explains how the signal waveform represented by the period P and the pulse width W translates to timer period and width values. Table 8-3 summarizes the register writes.

Table 8-3. Register Writes for Non-Overlapping Clock Pulses

Register	Before Enable	After Enable	At IRQ1	At IRQ2
TIMER5_PERIOD	P/2			
TIMER5_WIDTH	P/2 - W/2	W/2	P/2 - W/2	W/2
TIMER4_PERIOD	P	P/2		
TIMER4_WIDTH	P - W/2		W/2	P/2 - W-2

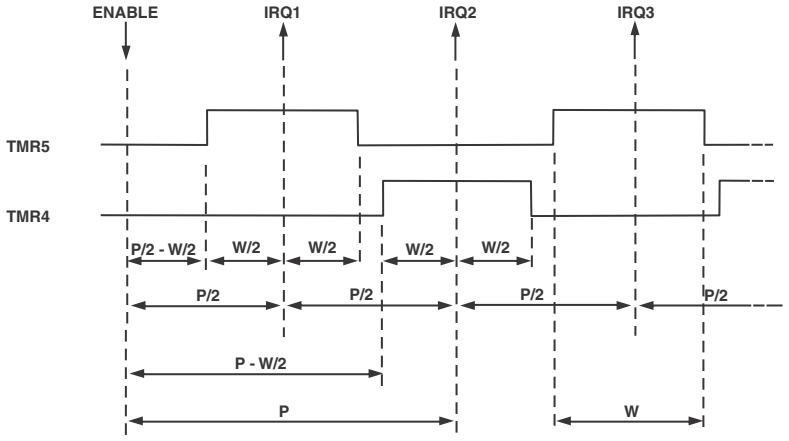


Figure 8-23. Non-Overlapping Clock Pulses

Since hardware only updates the written period and width values at the end of periods, software can write new values immediately after the timers have been enabled. Note that both timers' period expires at exactly the same times with the exception of the first timer 5 interrupt (at IRQ1) which is not visible to timer 4.

Listing 8-5. Non-Overlapping Clock Pulses

```
#define P 0x1000   /* signal period */
#define W 0x0600   /* signal pulse width */
#define N 4        /* number of pulses before disable */
timer45_toggle_hi:
    [--sp] = (r7:1, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* config timers */
    r7.l = IRQ_ENA | PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
/* calculate timers widths and period */
    r0.l = lo(P);
    r0.h = hi(P);
    r1.l = lo(W);
    r1.h = hi(W);
    r2 = r1 >> 1;    /* W/2 */
    r3 = r0 >> 1;    /* P/2 */
    r4 = r3 - r2;    /* P/2 - W/2 */
    r5 = r0 - r2;    /* P - W/2 */
/* write values for initial period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r0;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r5;
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r4;
```

```
/* start timers */
    r7.l = TIMEN5 | TIMEN4 ;
    w[p5 + TIMER_ENABLE - TIMER_ENABLE] = r7;
/* write values for second period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r2;
/* r0 functions as signal period counter */
    r0.h = hi(N * 2 - 1);
    r0.l = lo(N * 2 - 1);
    (r7:1, p5:5) = [sp++];
    rts;
timer45_toggle_hi.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:5, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* clear interrupt request */
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5 + TIMER_STATUS - TIMER_ENABLE] = r7;
/* toggle width values (width = period - width) */
    r7 = [p5 + TIMER5_PERIOD - TIMER_ENABLE];
    r6 = [p5 + TIMER5_WIDTH - TIMER_ENABLE];
    r5 = r7 - r6;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r5;
    r5 = [p5 + TIMER4_WIDTH - TIMER_ENABLE];
    r7 = r7 - r5;
    CC = r7 < 0;
    if CC r7 = r6;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
/* disable after a certain number of periods */
    r0+= -1;
    CC = r0 == 0;
```

```
    r5.l = 0;
    r7.l = TIMDIS5 | TIMDIS4;
    if !CC r7 = r5;
    w[p5 + TIMER_DISABLE - TIMER_ENABLE] = r7;
    (r7:5, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 8-5 generates N pulses on both timer output pins. Disabling the timers does not corrupt the generated pulse pattern anyhow.

Listing 8-6 configures timer 5 in `WDTH_CAP` mode. If looped back externally, this code might be used to receive N PWM patterns generated by one of the other timers. Ensure that the PWM generator and consumer both use the same `PERIOD_CNT` and `PULSE_HI` settings.

Listing 8-6. Timer Configured in WDTH_CAP Mode

```
.section L1_data_a;
.align 4;
#define N 1024
.var buffReceive[N*2];
.section L1_code;
timer5_capture:
    [--sp] = (r7:7, p5:5);
/* setup DAG2 */
    r7.h = hi(buffReceive);
    r7.l = lo(buffReceive);
    i2 = r7;
    b2 = r7;
    l2 = length(buffReceive)*4;
/* config timer for high pulses capture */
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
```

```
    r7.l = EMU_RUN|IRQ_ENA|PERIOD_CNT|PULSE_HI|WDTH_CAP;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = TIMEN5;
    w[p5 + TIMER_ENABLE - TIMER_ ENABLE] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer5_capture.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
/* clear interrupt request first */
    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r7 = [p5 + TIMER5_PERIOD - TIMER_STATUS];
    [i2++] = r7;
    r7 = [p5 + TIMER5_WIDTH - TIMER_STATUS];
    [i2++] = r7;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Unique Information for the ADSP-BF59x Processor

The ADSP-BF59x processor features one general-purpose timer module that contains three identical 32-bit timers. Each timer can be individually configured to operate in various modes. Although the timers operate com-

pletely independently of each other, all of them can be started and stopped simultaneously for synchronous operation.

Interface Overview

Figure 8-24 shows the ADSP-BF59x specific block diagram of the general-purpose timer module.

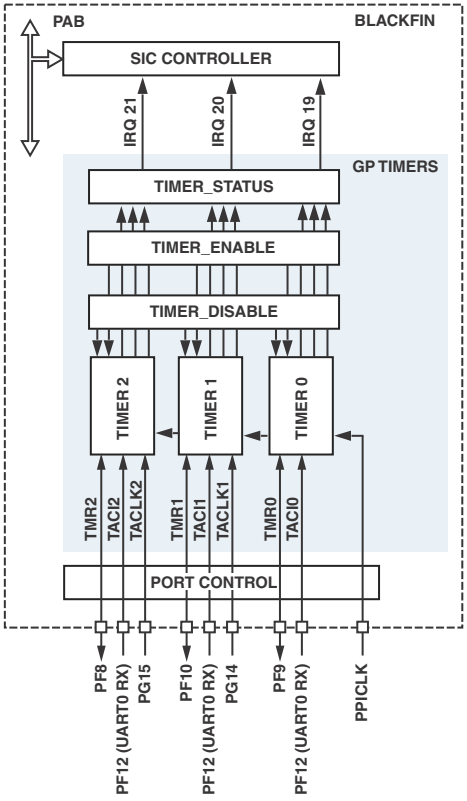


Figure 8-24. Timer Block Diagram

External Interface

The TMRCLK input is common to all three timers. The PPI unit is clocked by the same pin; therefore any of the timers can be clocked by PPI_CLK. Since timer 0 and timer 1 are often used in conjunction with the PPI, they are internally looped back to the PPI module for frame sync generation.

The timer signals TMR0 and TMR1 are multiplexed with the PPI frame syncs when the frame syncs are applied externally. PPI modes requiring only one frame sync free up TMR1. For details, see the *Parallel Peripheral Interface* chapter.

If the PPI frame syncs are applied externally, timer 0 and timer 1 are still fully functional and can be used for other purposes not involving the TMRx pins. Timer 0 and timer 1 must not drive their TMR0 and TMR1 pins. If operating in PWM_OUT mode, the OUT_DIS bit in the TIMER0_CONFIG and TIMER1_CONFIG registers must be set.

9 CORE TIMER

This chapter describes the core timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with a programming example.

Specific Information for the ADSP-BF59x

For details regarding the number of core timers for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For Core Timer interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

For a list of MMR addresses for each Core Timer, refer to Chapter A, "System MMR Assignments".

Core timer behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 9-9

Overview and Features

The core timer is a programmable 32-bit interval timer which can generate periodic interrupts. Unlike other peripherals, the core timer resides

inside the Blackfin core and runs at the core clock (CCLK) rate. Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operates at core clock (CCLK) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

Timer Overview

Figure 9-1 provides a block diagram of the core timer.

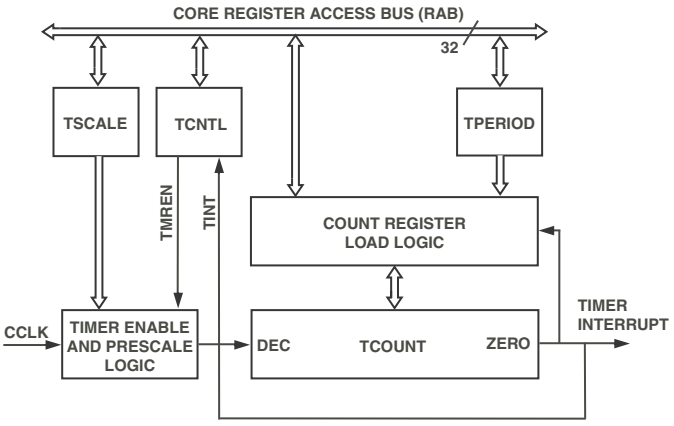


Figure 9-1. Core Timer Block Diagram

External Interfaces

The core timer does not directly interact with any pins of the chip.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus (RAB). The module is clocked by the core clock CCLK. The timer's dedicated interrupt request is a higher priority than requests from all other peripherals.

Description of Operation

The software should initialize the TCOUNT register *before* the timer is enabled. The TCOUNT register can be written directly, but writes to the TPERIOD register are also passed through to TCOUNT.

When the timer is enabled by setting the TMREN bit in the core timer control register (TCNTL), the TCOUNT register is decremented once every time the prescaler TSCALE expires, that is, every TSCALE + 1 number of CCLK clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register.

If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again. If the TAUTORLD bit is not set, the timer stops operation.

The core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR is set, the core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN is set when TMPWR = 0.

Interrupt Processing

The timer's dedicated interrupt request is a higher priority than requests from all other peripherals. The request goes directly to the core event controller (CEC) and does not pass through the system interrupt

controller (SIC). Therefore, the interrupt processing is also completely in the CCLK domain.

The core timer interrupt request is edge-sensitive and cleared by hardware automatically as soon as the interrupt is serviced.

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is *not* a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core time module doesn't provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

Core Timer Registers

The core timer includes four core memory-mapped registers, the timer control register (TCNTL), the timer count register (TCOUNT), the timer period register (TPERIOD), and the timer scale register (TSCALE). As with all core MMRs, these registers are always accessed by 32-bit read and write operations.

Core Timer Control Register (TCNTL)

The TCNTL register, shown in Figure 9-2, functions as control and status register.

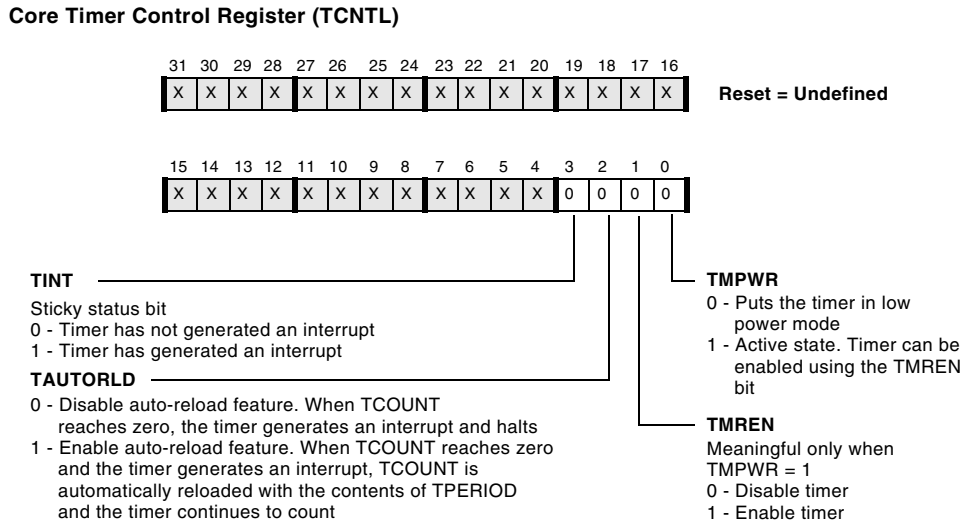

Core Timer Control Register (TCNTL)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X 0 0 0 0
TINT
Sticky status bit
0 - Timer has not generated an interrupt
1 - Timer has generated an interrupt
TAUTORLD
0 - Disable auto-reload feature. When TCOUNT reaches zero, the timer generates an interrupt and halts
1 - Enable auto-reload feature. When TCOUNT reaches zero and the timer generates an interrupt, TCOUNT is automatically reloaded with the contents of TPERIOD and the timer continues to count
TMPWR
0 - Puts the timer in low power mode
1 - Active state. Timer can be enabled using the TMREN bit
TMREN
Meaningful only when TMPWR = 1
0 - Disable timer
1 - Enable timer

Figure 9-2. Core Timer Control Register

Core Timer Count Register (TCOUNT)

The TCOUNT register, shown in Figure 9-3, decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

Values written to the TPERIOD register are automatically copied to the TCOUNT register. Nevertheless, the TCOUNT register can be written directly. In auto reload mode the value written to TCOUNT may differ from the TPERIOD value to let the initial period be shorter or longer than following periods. To do this, write to TPERIOD first and overwrite TCOUNT afterward.

Writes to TCOUNT are ignored once the timer is running.

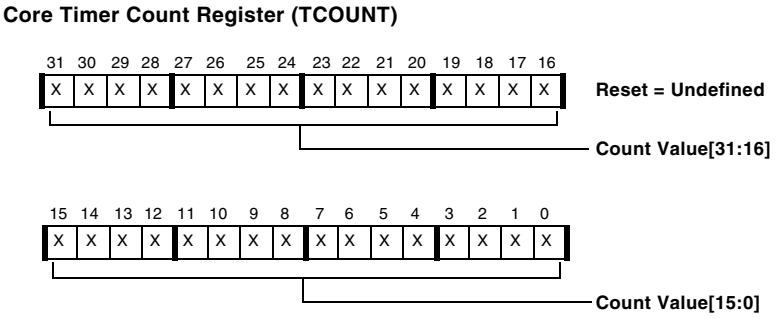

Core Timer Count Register (TCOUNT)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Count Value[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Count Value[15:0]

Figure 9-3. Core Timer Count Register

Core Timer Period Register (TPERIOD)

The TPERIOD register is shown in Figure 9-4. When auto-reload is enabled, the TCOUNT register is reloaded with the value of the TPERIOD register whenever TCOUNT reaches 0. Writes to TPERIOD are ignored when the timer is running.

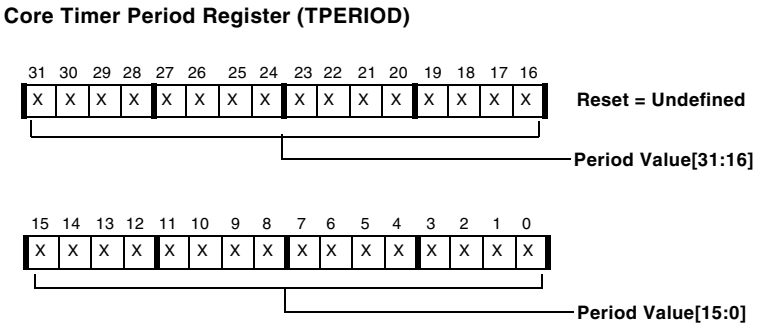

Core Timer Period Register (TPERIOD)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Period Value[31:16]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Period Value[15:0]

Figure 9-4. Core Timer Period Register

Core Timer Scale Register (TSCALE)

The TSCALE register is shown in Figure 9-5. The register stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every CCLK clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

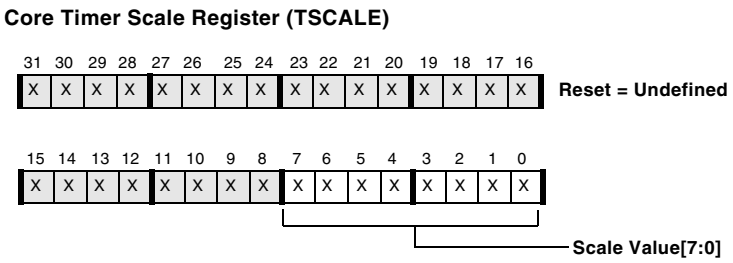

Core Timer Scale Register (TSCALE)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Scale Value[7:0]

Figure 9-5. Core Timer Scale Register

Programming Examples

Listing 9-1 configures the core timer in auto-reload mode. Assuming a CCLK of 500 MHz, the resulting period is 1 second. The initial period is twice as long as the others.

Listing 9-1. Core Timer Configuration

```
#include <defBF527.h>/*ADSP-BF527 product is used as an example*/
.section L1_code;
.global _main;
_main:
/* Register service routine at EVT6 and unmask interrupt */
    p1.l = lo(IMASK);
    p1.h = hi(IMASK);
```

```
    r0.l = lo(isr_core_timer);
    r0.h = hi(isr_core_timer);
    [p1 + EVT6 - IMASK] = r0;
    r0 = [p1];
    bitset(r0, bitpos(EVT_IVTMR));
    [p1] = r0;
/* Prescaler = 50, Period = 10,000,000, First Period = 20,000,000
*/
    p1.l = lo(TCNTL);
    p1.h = hi(TCNTL);
    r0 = 50 (z);
    [p1 + TSCALE - TCNTL] = r0;
    r0.l = lo(10000000);
    r0.h = hi(10000000);
    [p1 + TPERIOD - TCNTL] = r0;
    r0 <<= 1;
    [p1 + TCOUNT - TCNTL] = r0;
/* R6 counts interrupts */
    r6 = 0 (z);
/* start in auto-reload mode */
    r0 = TAUTORLD | TMPWR | TMREN (z);
    [p1] = r0;
_main.forever:
    jump _main.forever;
_main.end:
/* interrupt service routine simple increments R6 */
isr_core_timer:
    [--sp] = astat;
    r6+= 1;
    astat = [sp++];
    rti;
isr_core_timer.end:
```

Unique Information for the ADSP-BF59x Processor

None.

10 WATCHDOG TIMER

This chapter describes the watchdog timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of watchdog timers for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For Watchdog Timer interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

For a list of MMR addresses for each Watchdog Timer, refer to Chapter A, "System MMR Assignments".

Watchdog timer behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 10-10

Overview and Features

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the watchdog expires before being updated by software.

Watchdog timer key features include:

- 32-bit watchdog timer
- 8-bit disable bit pattern
- System reset on expire option
- NMI on expire option
- General-purpose interrupt option

Typically, the watchdog timer is used to supervise stability of the system software. When used in this way, software reloads the watchdog timer in a regular manner so that the downward counting timer never expires (never becomes 0). An expiring timer then indicates that system software might be out of control. At this point a special error handler may recover the system. For safety, however, it is often better to reset and reboot the system directly by hardware control.

Especially in slave boot configurations, a processor reset cannot automatically force the Blackfin device to be rebooted. In this case, the processor may reset without booting again and may negotiate with the host device by the time program execution starts. Alternatively, a watchdog event can cause an NMI event. The NMI service routine may request the host device reset and/or reboot the Blackfin processor.

The watchdog timer is often programmed to let the processor wake up from sleep mode after a programmable period of time.

For easier debugging, the watchdog timer does not decrement (even if enabled) when the processor is in emulation mode.

Interface Overview

Figure 10-1 provides a block diagram of the watchdog timer.

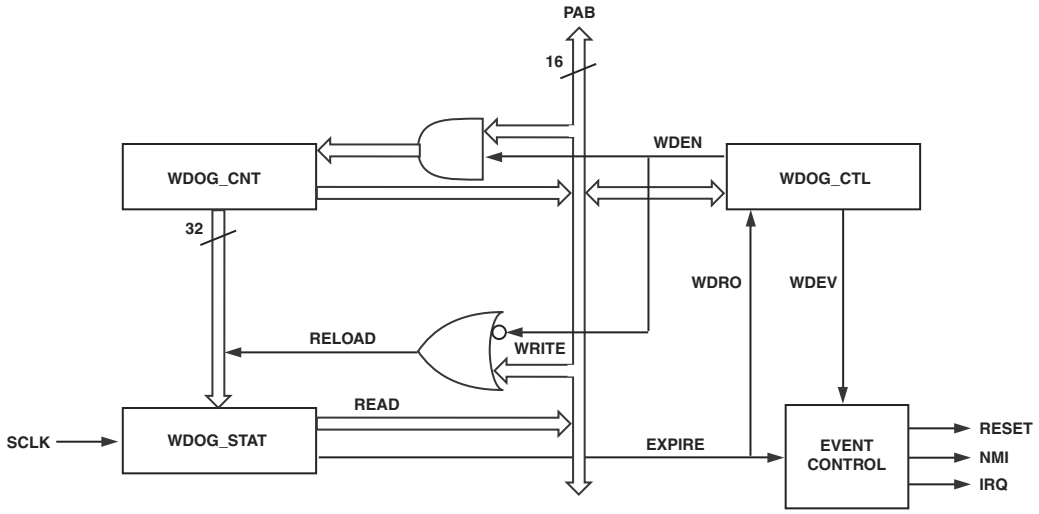


Figure 10-1. Watchdog Timer Block Diagram

External Interface

The watchdog timer does not directly interact with any pins of the chip.

Internal Interface

The watchdog timer is clocked by the system clock `SCLK`. Its registers are accessed through the 16-bit peripheral access bus (PAB). The 32-bit registers `WDOG_CNT` and `WDOG_STAT` must always be accessed by 32-bit read/write operations. Hardware ensures that those accesses are atomic.

When the counter expires, one of three event requests can be generated. Either a reset or an NMI request is issued to the core event controller

(CEC) or a general-purpose interrupt request is passed to the system interrupt controller (SIC).

Description of Operation

If enabled, the 32-bit watchdog timer counts downward every SCLK cycle. If it becomes 0, one of three event requests can be issued to either the CEC or the SIC. Depending on how the WDEV bit field in the WDOG_CTL register is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt.

The counter value can be read through the 32-bit WDOG_STAT register. The WDOG_STAT register cannot, however, be written directly. Rather, software writes the watchdog period value into the 32-bit WDOG_CNT register *before* the watchdog is enabled. Once the watchdog is started, the period value cannot be altered.

To start the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the watchdog count register (WDOG_CNT). Since the watchdog timer is not enabled yet, the write to the WDOG_CNT registers automatically pre-loads the WDOG_STAT register as well.
2. In the watchdog control register (WDOG_CTL), select the event to be generated upon timeout.
3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register.

If software does not service the watchdog in time, WDOG_STAT continues decrementing until it reaches 0. Then, the programmed event is generated. The counter stops decrementing and remains at zero. Additionally, the WDRO latch bit in the WDOG_CTL register is set and can be interrogated by software in case event generation is not enabled.

When the watchdog is programmed to generate a reset, it resets the processor core and peripherals. If the NOBOOT bit in the SYSCR register was set by the time the watchdog resets the part, the chip is not rebooted. This is recommended behavior in slave boot configurations. The reset handler may evaluate the RESET_WDOG bit in the software reset register SWRST to detect a reset caused by the watchdog. For details, see the *System Reset and Booting* chapter.

To prevent the watchdog from expiring, software services the watchdog by performing dummy writes to the WDOG_STAT register. The values written are ignored, but the write commands cause the WDOG_STAT register to be reloaded from the WDOG_CNT register.

If the watchdog is enabled with a zero value loaded to the counter and the WDRO bit was cleared, the WDRO bit of the watchdog control register is set immediately and the counter remains at zero without further decrements. If, however, the WDRO bit was set by the time the watchdog is enabled, the counter decrements to 0xFFFF FFFF and continues operation.

Software can disable the watchdog timer only by writing a 0xAD value to the WDEN field in the WDOG_CTL register.

Register Definitions

The watchdog timer is controlled by three registers.

Watchdog Count (WDOG_CNT) Register

The WDOG_CNT register, shown in Figure 10-2, holds the 32-bit unsigned count value. The WDOG_CNT register must always be accessed with 32-bit read/writes.

A valid write to the WDOG_CNT register also preloads the watchdog counter. For added safety, the WDOG_CNT register can be updated only when the

watchdog timer is disabled. A write to the WDOG_CNT register while the timer is enabled does not modify the contents of this register.

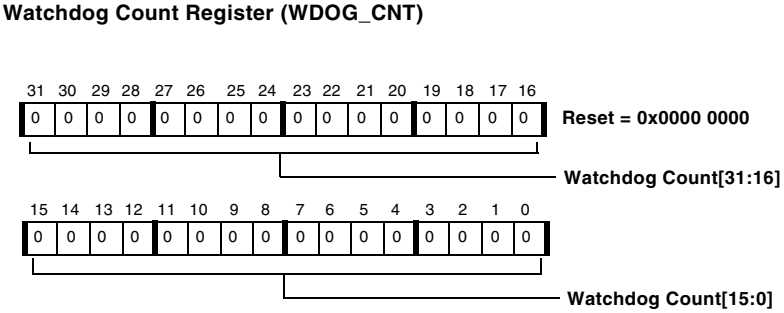


Figure 10-2. Watchdog Count Register

Watchdog Status (WDOG_STAT) Register

The 32-bit WDOG_STAT register, shown in Figure 10-3, contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways.

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.
- While the watchdog timer is enabled, but not rolled over yet, writes to the WDOG_STAT register load it with the value in WDOG_CNT.

Enabling the watchdog timer does not automatically reload WDOG_STAT from WDOG_CNT.

The WDOG_STAT register is a 32-bit unsigned system MMR that must be accessed with 32-bit reads and writes.

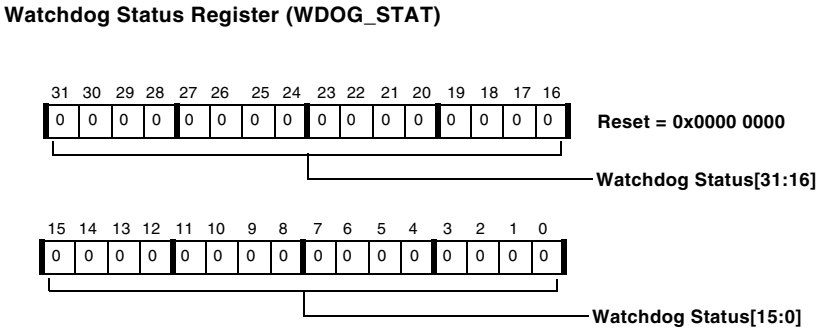


Figure 10-3. Watchdog Status Register

Watchdog Control (WDOG_CTL) Register

The WDOG_CTL register, shown in Figure 10-4, is a 16-bit system MMR used to control the watchdog timer.

The watchdog event (WDEV[1:0]) bit field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the SIC_IMASK register that holds the watchdog timer mask bit should be appropriately configured to unmask that interrupt. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The watchdog enable (WDEN[7:0]) bit field is used to enable and disable the watchdog timer. Writing any value other than the disable key (0xAD) into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.

Software can determine whether the watchdog has expired by interrogating the WDRO status bit of the WDOG_CTL register. This is a sticky bit that is set whenever the watchdog timer count reaches 0. It can be cleared only by writing a “1” to the bit when the watchdog has been disabled first.

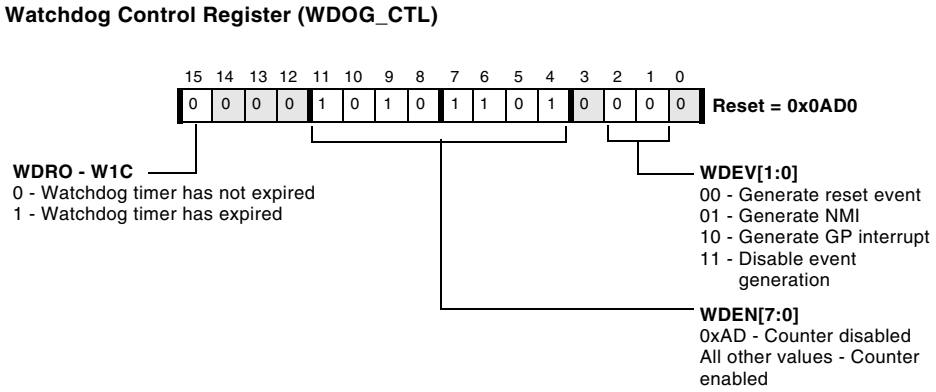

Watchdog Control Register (WDOG_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0
Reset = 0x0AD0
WDRO - W1C
0 - Watchdog timer has not expired
1 - Watchdog timer has expired
WDEV[1:0]
00 - Generate reset event
01 - Generate NMI
10 - Generate GP interrupt
11 - Disable event generation
WDEN[7:0]
0xAD - Counter disabled
All other values - Counter enabled

Figure 10-4. Watchdog Control Register

Programming Examples

Listing 10-1 shows how to configure the watchdog timer so that it resets the chip when it expires. At startup, the code evaluates whether the recent reset event has been caused by the watchdog. Additionally, the example sets the NOBOOT bit to prevent the memory from being rebooted.

Listing 10-1. Watchdog Timer Configuration

```
#include <defBF527.h>/*ADSP-BF527 product is used as an example*/
#define WDOGPERIOD 0x00200000
.section L1_code;
.global _reset;
_reset:
  ...
/* optionally, test whether reset was caused by watchdog */
    p0.h=hi(SWRST);
    p0.l=lo(SWRST);
    r6 = w[p0] (z);
```

```
    CC = bittst(r6, bitpos(RESET_WDOG));
    if !CC jump _reset.no_watchdog_reset;
/* optionally, warn at system level or host device here */
_reset.no_watchdog_reset:
/* optionally, set NOBOOT bit to avoid reboot in case */
    p0.h=hi(SYSCR);
    p0.l=lo(SYSCR);
    r0 = w[p0](z);
    bitset(r0,bitpos(NOBOOT));
    w[p0] = r0;

/* start watchdog timer, reset if expires */
    p0.h = hi(WDOG_CNT);
    p0.l = lo(WDOG_CNT);
    r0.h = hi(WDOGPERIOD);
    r0.l = lo(WDOGPERIOD);
    [p0] = r0;
    p0.l = lo(WDOG_CTL);
    r0.l = WDEN | WDEV_RESET;
    w[p0] = r0;
    ...
    jump _main;
_reset.end:
```

The subroutine shown in Listing 10-2 can be called by software to service the watchdog. Note that the value written to the `WDOG_STAT` register does not matter.

Listing 10-2. Service Watchdog

```
service_watchdog:
    [--sp] = p5;
    p5.h = hi(WDOG_STAT);
    p5.l = lo(WDOG_STAT);
    [p5] = r0;
```

```
    p5 = [sp++];
    rts;
service_watchdog.end:
```

Listing 10-3 is an interrupt service routine that restarts the watchdog. Note that the watchdog must be disabled first.

Listing 10-3. Watchdog Restarted by Interrupt Service Routine

```
isr_watchdog:
    [--sp] = astat;
    [--sp] = (p5:5, r7:7);
    p5.h = hi(WDOG_CTL);
    p5.l = lo(WDOG_CTL);
    r7.l = WDDIS;
    w[p5] = r7;
    bitset(r7, bitpos(WDRO));
    w[p5] = r7;
    r7 = [p5 + WDOG_CNT - WDOG_CTL];
    [p5 + WDOG_CNT - WDOG_CTL] = r7;
    r7.l = WDEN | WDEV_GPI;
    w[p5] = r7;
    (p5:5, r7:7) = [sp++];
    astat = [sp++];
    rti;
isr_watchdog.end:
```

Unique Information for the ADSP-BF59x Processor

None.

11 UART PORT CONTROLLERS

This chapter describes the universal asynchronous receiver/transmitter (UART) module. Following an overview and a list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of UARTs for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For UART DMA channel assignments, refer to Table 5-9 on page 5-103 in Chapter 5, "Direct Memory Access".

For UART interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

To determine how each of the UARTs is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each UART, refer to Chapter A, "System MMR Assignments".

UART behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 11-41

Overview

The UART module is a full-duplex peripheral compatible with PC-style industry-standard UARTs, sometimes called serial controller interfaces (SCI). UARTs convert data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, bit rate, and parity generation options.

Features

Each UART includes these features:

- 5 – 8 data bits
- 1 or 2 stop bits (1½ in 5-bit mode)
- Even, odd, and sticky parity bit options
- 3 interrupt outputs for reception, transmission, and status
- Independent DMA operation for receive and transmit
- SIR IrDA operation mode
- Internal loop back

The UART is logically compliant to EIA-232E, EIA-422, EIA-485 and LIN standards, but usually requires an external transceiver device to meet electrical requirements. In IrDA® (Infrared Data Association) mode, the UART meets the half-duplex IrDA SIR (9.6/115.2 Kbps rate) protocol.

Interface Overview

Figure 11-1 on page 11-3 shows a simplified block diagram of a UART module and how it interconnects to the Blackfin architecture and to the outside world.

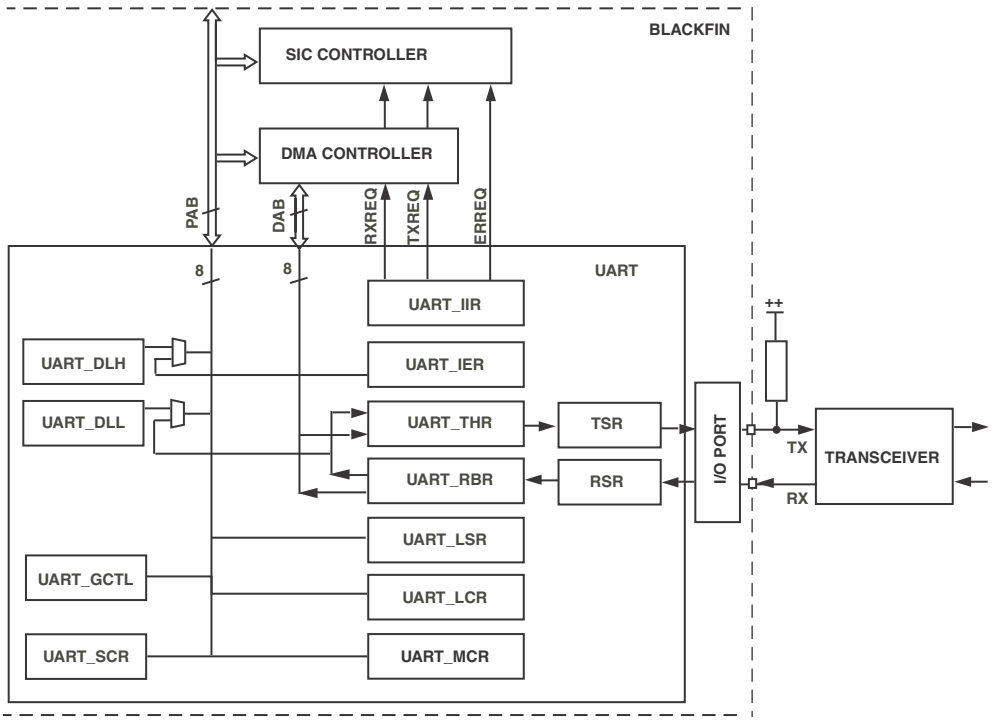


Figure 11-1. UART Block Diagram

External Interface

Each UART features an RX and a TX pin. These two pins usually connect to an external transceiver device that meets the electrical requirements of

full duplex (for example, EIA-232, EIA-422, 4-wire EIA-485) or half duplex (for example, 2-wire EIA-485, LIN) standards.

The RX and TX pins do not need to be used together. If only receive or transmit functionality of a UART module is needed, the unused pin may be used for an alternate function, depending on the port multiplexing scheme of a specific processor. For more details on functionality multiplexed with the UART pins, see Chapter 7, "General-Purpose Ports".

Modem status and control functionality is not supported by the UART modules, but may be implemented using GPIO pins.

Internal Interface

UARTs are DMA-capable peripherals with support for separate TX and RX DMA master channels. They can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. Each UART has its own separate transmit and receive DMA channels. For more information on DMA, see the *Direct Memory Access* chapter.

All UART registers are eight bits wide. They connect to the peripheral bus. However, some registers share their address as controlled by the `DLAB` bit in the `UART_LCR` register. The `UART_RBR` and `UART_THR` registers also connect to the DAB bus

A hardware-assisted autobaud detection mechanism is accomplished by coupling a specific GP Timer with a specific UART. For information on GP Timer - UART pairings for autobaud detection, see General-Purpose Ports chapter.

Description of Operation

The following sections describe the operation of the UART.

UART Transfer Protocol

UART communication follows an asynchronous serial protocol, consisting of individual data words. A word has 5 to 8 data bits.

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the line control register (`UART_LCR`). Data is always transmitted and received least significant bit (LSB) first.

Figure 11-2 on page 11-6 shows a typical physical bitstream measured on one of the TX pins.

Aside from the standard UART functionality, the UART also supports half-duplex serial data communication via infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16× data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16× clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

IrDA support is enabled by setting the `IREN` bit in the `UART_GCTL` register. The IrDA application requires external transceivers.

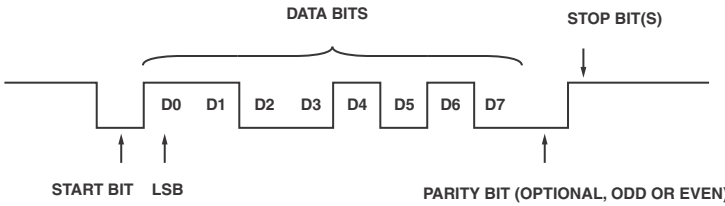


Figure 11-2. Bitstream on a TX Pin Transmitting an “S” Character (0x53)

UART Transmit Operation

Receive and transmit paths operate independently except that the bit rate and the frame format are identical for both transfer directions.

Transmission is initiated by writes to the UART_THR register. If no former operation is pending, the data is immediately passed from the UART_THR register to the internal TSR register where it is shifted out at a bit rate equal to SCLK/(16 × Divisor) (see “Bit Rate Generation” on page 11-12 for information about the divisor) with start, stop, and parity bits appended as defined the UART_LCR register. The least significant bit (LSB) is always transmitted first. This is bit 0 of the value written to UART_THR.

Writes to the UART_THR register clear the THRE flag. Transfers of data from UART_THR to the transmit shift registers (TSR) set this status flag in UART_LSR again.

When enabled by the ETBEI bit in the UART_IER register, a 0 to 1 transition of the THRE flag requests an interrupt on the dedicated TXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the TXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the system interrupt controller (SIC).

The UART_THR register and the internal TSR register can be seen as a two-stage transmit buffer. When data is pending in either one of these

registers, the TEMT flag is low. As soon as the data has left the TSR register, the TEMT bit goes high again and indicates that all pending transmit operation has finished. At that time it is safe to disable the UCEN bit or to three-state off-chip line drivers.

UART Receive Operation

The receive operation uses the same data format as the transmit configuration, except that one valid stop bit is always sufficient. That is, the STB bit has no impact to the receiver.

After detection of the start bit, the received word is shifted into the internal shift register (RSR) at a bit rate of SCLK/(16 × Divisor) . Once the appropriate number of bits (including one stop bit) is received, the content of the RSR register is transferred to the UART_RBR registers, shown in Figure 11-11 on page 11-26. Finally, the data ready (DR) bit and the status flags are updated in the UART_LSR register, to signal data reception, parity, and also error conditions, if required.

The RSR and the UART_RBR registers can be seen as almost a two-stage receive buffer. If the stop bit of a second byte is received before software reads the first byte from the UART_RBR register, an overrun error is reported and the first byte is overwritten.

If enabled by the ERBFI bit in the UART_IER register, a 0 to 1 transition of the DR flag requests an interrupt on the dedicated RXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the RXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the system interrupt controller.

If errors are detected during reception, an interrupt can be requested to a separate error interrupt output. This error request goes directly to the system interrupt controller. However, it is hard-wired with the error requests of other modules. The error handler routine may need to interrogate multiple modules as to whether they requested the event. Error requests must be enabled by the ELSI bit in the UART_IER register. The following error

situations are detected. Every error has an indicating bit in the UART_LSR register.

- Overrun error (OE bit)
- Parity error (PE bit)
- Framing error/Invalid stop bit (FE bit)
- Break indicator (BI bit)

Reception is started when a falling edge is detected on the RX input pin. The receiver attempts to see a start bit. For better immunity against noise and hazards on the line, the receiver oversamples every bit 16 times and does a majority decision based on the middle three samples. The data is shifted immediately into the internal RSR register. After the 9th sample of the first stop bit is processed, the received data is copied to the UART_RBR register and the receiver recovers itself for further data.

The sampling clock, equal to 16 times the bit rate, samples the data bits close to their midpoint. Because the receiver clock is usually asynchronous to the transmitter's data rate, the sampling point may drift relative to the center of the data bits. The sampling point is synchronized again with each start bit, so the error accumulates only over the length of a single word. A receive filter removes spurious pulses of less than two times the sampling clock period.

IrDA Transmit Operation

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted if the TPOLC bit is cleared, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3 clock

periods out of 16 clock periods in the cycle. The pulse is centered around the middle of the bit time, as shown in Figure 11-3. The final IrDA pulse is fed to the off-chip infrared driver.

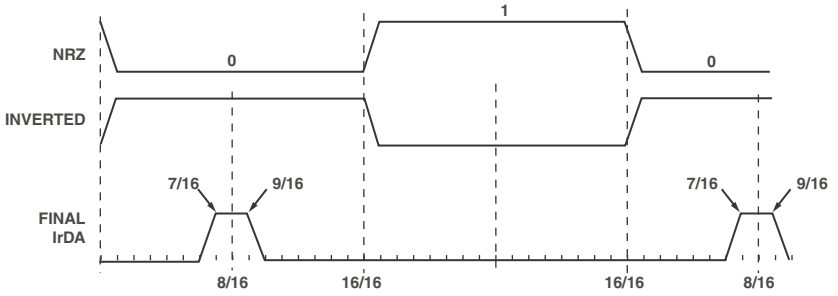


Figure 11-3. IrDA Transmit Pulse

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 11-1 on page 11-13, the error terms associated with the bit rate generator are very small and well within the tolerance of most infrared transceiver specifications.

IrDA Receive Operation

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.

Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the

transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note that because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16× bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the IRPOL bit. Figure 11-4 on page 11-10 gives examples of each polarity type.

- IRPOL = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- IRPOL = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.

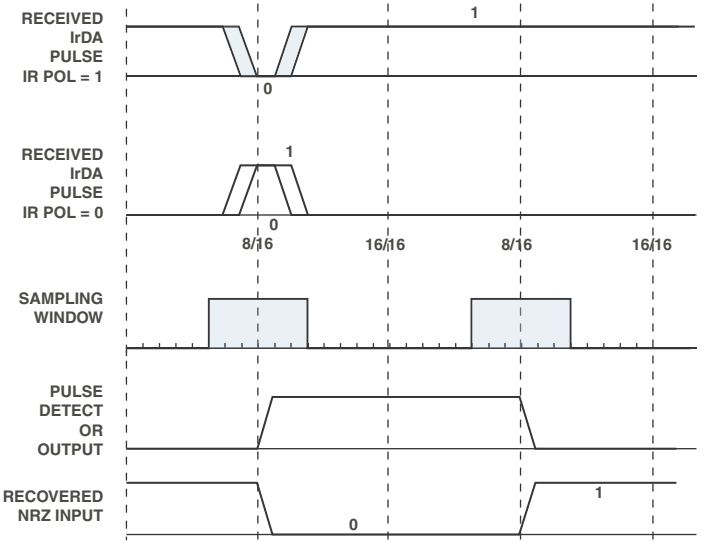


Figure 11-4. IrDA Receiver Pulse Detection

Interrupt Processing

Each UART module has three interrupt outputs. One is dedicated for transmission, one for reception, and the third is used to report line status. As shown in Figure 11-1 on page 11-3, the transmit and receive requests are routed through the DMA controller. The status request goes directly to the system interrupt controller after being ORed with interrupt signals from other modules.

If the associated DMA channel is enabled, the request functions as a DMA request. If the DMA channel is disabled, it simply forwards the request to the system interrupt controller. Note that a DMA channel must be associated with the UART module to enable TX and RX interrupts. Otherwise, the transmit and receive requests cannot be forwarded. Refer to the description of the peripheral map registers in the *Direct Memory Access* chapter.

Transmit interrupts are enabled by the ETBEI bit in the UART_IER register. If set, the transmit request is asserted when the THRE bit in the UART_LSR register transitions from 0 to 1, indicating that the TX buffer is ready for new data.

Note that the THRE bit resets to 1. When the ETBEI bit is set in the UART_IER register, the UART module immediately issues an interrupt or DMA request. In this way, no special handling of the first character is required when transmission of a string is initiated. Simply set the ETBEI bit and let the interrupt service routine load the first character from memory and write it to the UART_THR register in the normal manner. Accordingly, the ETBEI bit can be cleared if the string transmission has completed. For more information, see “DMA Mode” on page 11-17.

The THRE bit is cleared by hardware when new data is written to the UART_THR register. These writes also clear the TX interrupt request. However, they also initiate further transmission. If software doesn’t want to continue transmission, the TX request can alternatively be cleared by either clearing the ETBEI bit or by reading the UART_IIR register.

Receive interrupts are enabled by the ERBFI bit in the UART_IER register. If set, the receive request is asserted when the DR bit in the UART_LSR register transitions from 0 to 1, indicating that new data is available in the UART_RBR register. When software reads the UART_RBR, hardware clears the DR bit again. Reading UART_RBR also clears the RX interrupt request.

Status interrupts are enabled by the ELSI bit in the UART_IER register. If set, the status interrupt request is asserted when any error bit in the UART_LSR register transitions from 0 to 1. Refer to "UART Line Status (UART_LSR) Register" on page 11-24 for details. Reading the UART_LSR register clears the error bits destructively. These reads also clear the status interrupt request.

For legacy reasons, the UART_IIR registers still reflect the UART interrupt status. Legacy operation may require bundling all UART interrupt sources to a single interrupt channel and servicing them all by the same software routine. This can be established by globally assigning all UART interrupts to the same interrupt priority, by using the system interrupt controller.

If either the line status interrupt or the receive data interrupt has been assigned a lower interrupt priority by the system interrupt controller, a deadlock condition can occur. To avoid this, always assign the lowest priority of the enabled UART interrupts to the UART_THR empty event.

Bit Rate Generation

The UART clock is enabled by the UCEN bit in the UART_GCTL register.

The bit rate is characterized by the system clock (SCLK) and the 16-bit divisor. The divisor is split into the UART_DLL and the UART_DLH registers. These registers form a 16-bit divisor. The bit clock is divided by 16 so that:

bit rate = SCLK/(16 × divisor)
divisor = 65536 when UART_DLL = UART_DLH = 0

Table 11-1 provides example divide factors required to support most standard baud rates.

Table 11-1. UART Bit Rate Examples With 100 MHz SCLK

Bit Rate	DL	Actual	% Error
2400	2604	2400.15	0.006
4800	1302	4800.31	0.007
9600	651	9600.61	0.006
19200	326	19171.78	0.147
38400	163	38343.56	0.147
57600	109	57339.45	0.452
115200	54	115740.74	0.469
921600	7	892857.14	3.119
6250000	1	6250000	–

Careful selection of SCLK frequencies, that is, even multiples of desired bit rates, can result in lower error percentages.

Note that the UART module is clocked 16 times faster than the bit clock. This is required to oversample bits on reception and to generate RZI code in IrDA mode.

Autobaud Detection

At the chip level, the UART RX pin is routed to the alternate capture input (TACIx) of a general purpose timer. When working in WDTH_CAP mode this timer can be used to automatically detect the bit rate applied to the RX pin by an external device. For more information, see Chapter 7, “General-Purpose Ports”.

The capture capabilities of the timers are often used to supervise the bit rate at runtime. If the Blackfin UART talks to a device supplied by a weak

clock oscillator that drifts over time, the Blackfin can re-adjust its UART bit rate dynamically.

Often, autobaud detection is used for initial bit rate negotiations. In this case, the Blackfin processor is most likely a slave device waiting for the host to send a predefined autobaud character (see below). This is the scenario used for UART booting. In this scenario, the UART clock enable bit UCEN should not be enabled while autobaud detection is performed. This prevents the UART from starting reception with incorrect bit rate matching. Alternatively, the UART can be disconnected from the RX pin by setting the LOOP_ENA bit.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with the UART operation—all derived from SCLK—the pulse widths can be used to calculate the baud rate divider for the UART.

divisor = TIMER_WIDTH/(16 × number of captured UART bits)

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more bits. Traditionally, a NULL character (ASCII 0x00) was used in autobaud detection, as shown in Figure 11-5.

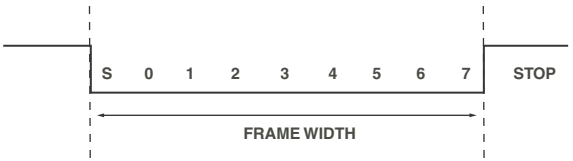


Figure 11-5. Autobaud Detection Character 0x00

Because the example frame in Figure 11-5 encloses 8 data bits and 1 start bit, apply the formula:

divisor = TIMER_WIDTH/(16 × 9)

Real UART RX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character "@" (0x40) as the autobaud detection character and measure the period between two subsequent falling edges. As shown in Figure 11-6, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses eight bits, apply the formula:

divisor = `TIMER_PERIOD`>>7

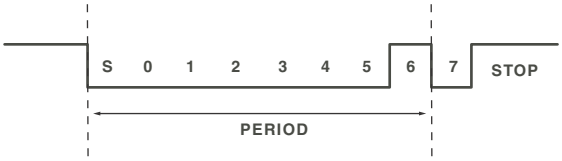


Figure 11-6. Autobaud Detection Character 0x40

An example is provided in Listing 11-2 on page 11-33.

Programming Model

The following sections describe a programming model for the UART.

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into `UART_THR`. Received data can be read from `UART_RBR`. The processor must write and read one character at time.

To prevent any loss of data and misalignments of the serial datastream, the UART_LSR register provides two status flags for handshaking—THRE and DR.

The THRE flag is set when UART_THR is ready for new data and cleared when the processor loads new data into UART_THR. Writing UART_THR when it is not empty overwrites the register with the new value and the previous character is never transmitted.

The DR flag signals when new data is available in UART_RBR. This flag is cleared automatically when the processor reads from UART_RBR. Reading UART_RBR when it is not full returns the previously received value. When UART_RBR is not read in time, newly received data overwrites UART_RBR and the OE flag is set.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Be careful if transmit and receive are served by different software threads, because read operations on the UART_LSR and UART_IIR registers are destructive. Polling the SIC_ISR register without enabling the interrupts by SIC_MASK is an alternate method of operation to consider. Software can write up to two words into the UART_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines. Separate interrupt lines are provided for UART TX, UART RX, and UART error/status. The independent interrupts can be enabled individually by the UART_IER register.

The ISRs can evaluate the status bit field within the UART_IIR register to determine the signalling interrupt source. If more than one source is signalling, the status field displays the one with the highest priority. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 11-13 on page 11-29.

DMA Mode

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The software does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through autobuffer mode.

DMA channels provide a 4-deep FIFO, resulting in total buffer capabilities of 6 words at both the transmit and receive sides. In DMA mode, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities.

DMA interrupt routines must explicitly write "1" to the corresponding DMA_IRQ_STATUS registers to clear the latched request of the pending interrupt.

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UART_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. The UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

For transmit DMA, it is recommended that the SYNC bit in the DMA_CONFIG register be set. With this bit set, the interrupt generation is delayed until the entire DMA FIFO has been drained to the UART module. The UART TX DMA interrupt service routine is allowed to start another DMA sequence or to clear the ETBEI control bit only when the SYNC bit is set.

If another DMA is started while data is still pending in the UART transmitter, there is no need to pulse the ETBEI bit to initiate the second DMA. If, however, the recent byte has already been loaded into the TSR registers

(that is, the THRE bit is set), then the ETBEI bit must be cleared and set again to let the second DMA start.

In DMA transmit mode, the ETBEI bit enables the peripheral request to the DMA FIFO. The strobe on the memory side is still enabled by the DMAEN bit. If the DMA count is less than the DMA FIFO depth, which is 4, then the DMA interrupt might be requested before the ETBEI bit is set. If this is not wanted, set the SYNC bit in the DMA_CONFIG register.

Regardless of the SYNC setting, the DMA stream has not left the UART transmitter completely at the time the interrupt is generated. If the UART clock was disabled without additional polling of the TEMT bit, transmission may abort in the middle of the stream—causing data loss.

The UART's DMA supports 8-bit and 16-bit operation, but not 32-bit operation. Sign extension is not supported.

Mixing Modes

Especially on the transmit side, switching from DMA mode to non-DMA operation on the fly requires some thought. By default, the interrupt timing of the DMA is synchronized with the memory side of the DMA FIFOs. The TX DMA completion interrupt is generated after the last byte has been copied from the memory into the DMA FIFO. The TX DMA interrupt service routine is not yet permitted to start other DMA sequences or to switch to non-DMA transmission. The interrupt is requested by the time the DMA_DONE bit is set. The DMA_RUN bit, however, remains set until the data has completely left the TX DMA FIFO.

Therefore, when planning to switch from DMA to non-DMA of operation, always set the SYNC bit in the DMA_CONFIG word of the last descriptor or work unit before handing over control to non-DMA mode. Then, after the interrupt occurs, software can write new data into the UART_THR register as soon as the THRE bit permits. If the SYNC bit cannot be set, software can poll the DMA_RUN bit instead.

When switching from non-DMA to DMA operation, take care that the very first DMA request is issued properly. If the DMA is enabled while the UART is still transmitting, no precaution is required. If, however, the DMA is enabled after the TEMT bit became high, the ETBEI bit should be pulsed to initiate DMA transmission.

UART Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These memory-mapped registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero filled. Table 11-2 on page 11-19 provides an overview of the UART registers.

Consistent with industry-standard devices, multiple registers are mapped to the same address location. The UART_DLH and UART_DLL registers share their addresses with the UART_THR registers, the UART_RBR registers, and the UART_IER registers. The DLAB bit in the UART_LCR register controls which set of registers is accessible at a given time. Software must use 16-bit word load/store instructions to access these registers.

Transmit and receive channels are both buffered. The UART_THR registers buffer the transmit shift register (TSR) and the UART_RBR registers buffer the receive shift register (LSR). The shift registers are not directly accessible by software.

Table 11-2. UART Register Overview

Name	Address Offset	DLAB Bit Setting	Operation	Reset Value	Function
UART_RBR	0x0000	0	R	0x00	Receive buffer register
UART_THR	0x0000	0	W	0x00	Transmit holding register
UART_DLL	0x0000	1	R/W	0x01	Divisor latch low byte

Table 11-2. UART Register Overview

Name	Address Offset	DLAB Bit Setting	Operation	Reset Value	Function
UART_IER	0x0004	0	R/W	0x00	Interrupt enable register
UART_DLH	0x0004	1	R/W	0x00	Divisor latch high byte
UART_IIR	0x0008	X	R Read operations are destructive	0x01	Interrupt identification register
UART_LCR	0x000C	X	R/W	0x00	Line control register
UART_MCR	0x0010	X	R/W	0x00	Modem control register
UART_LSR	0x0014	X	R Read operations are destructive	0x60	Line status register
UART_SCR	0x001C	X	R/W	0x00	Scratch register
UART_GCTL	0x0024	X	R/W	0x00	Global control register

UART Line Control (UART_LCR) Register

The UART_LCR register, shown in Figure 11-7, controls the format of received and transmitted character frames.

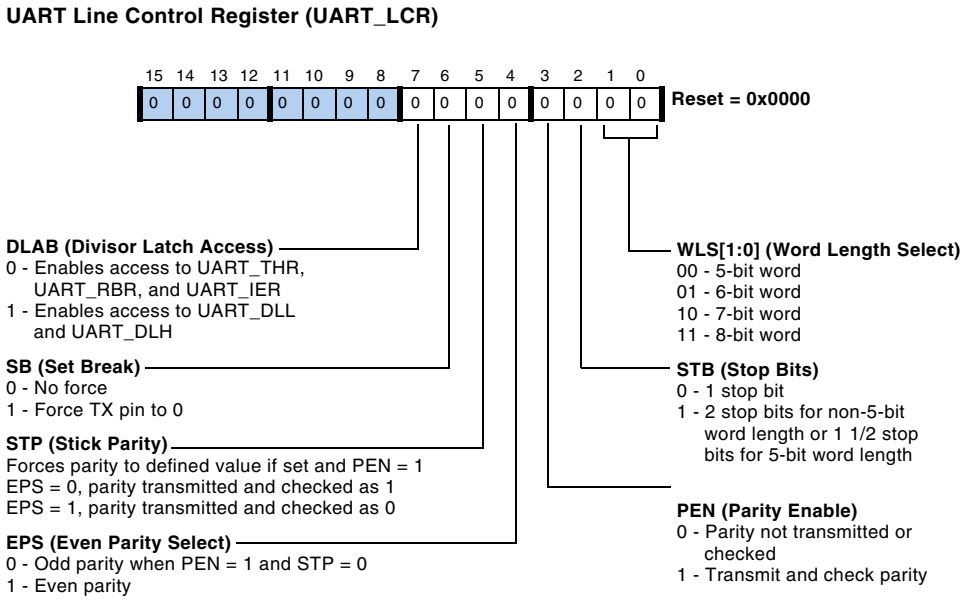

UART Line Control Register (UART_LCR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
DLAB (Divisor Latch Access)
0 - Enables access to UART_THR, UART_RBR, and UART_IER
1 - Enables access to UART_DLL and UART_DLH
SB (Set Break)
0 - No force
1 - Force TX pin to 0
STP (Stick Parity)
Forces parity to defined value if set and PEN = 1
EPS = 0, parity transmitted and checked as 1
EPS = 1, parity transmitted and checked as 0
EPS (Even Parity Select)
0 - Odd parity when PEN = 1 and STP = 0
1 - Even parity
WLS[1:0] (Word Length Select)
00 - 5-bit word
01 - 6-bit word
10 - 7-bit word
11 - 8-bit word
STB (Stop Bits)
0 - 1 stop bit
1 - 2 stop bits for non-5-bit word length or 1 1/2 stop bits for 5-bit word length
PEN (Parity Enable)
0 - Parity not transmitted or checked
1 - Transmit and check parity

Figure 11-7. UART Line Control Register

The 2-bit WLS field determines whether the transmitted and received UART word consists of 5, 6, 7 or 8 data bits.

The STB bit controls how many stop bits are appended to transmitted data. When STB = 0, one stop bit is transmitted. If WLS is non zero, STB = 1 instructs the transmitter to add one additional stop bit, two stop bits in total. If WLS = 0 and 5-bit operation is chosen, STB = 1 forces the transmitter to append one additional half bit, 1½ stop bits in total. Note that this bit does not impact data reception—the receiver is always satisfied with one stop bit.

The PEN bit inserts one additional bit between the most significant data bit and the first stop bit. The polarity of this so-called parity bit depends on data and the STP and EPS control bits. Both transmitter and receiver calculate the parity value. The receiver compares the received parity bit with the expected value and issues a parity error if they don't match. If PEN is cleared, the STP and the EPS bits are ignored.

The STP bit controls whether the parity is generated by hardware based on the data bits or whether it is set to a fixed value. If STP = 0 the hardware calculates the parity bit value based on the data bits. Then, the EPS bit determines whether odd or even parity mode is chosen. If EPS = 0, odd parity is used. That means that the total count of logical-1 data bits including the parity bit must be an odd value. Even parity is chosen by STP = 0 and EPS = 1. Then, the count of logical-1 bits must be a even value. If the STP bit is set, then hardware parity calculation is disabled. In this case, the sent and received parity equals the inverted EPS bit. The example in Table 11-3 summarizes polarity behavior assuming 8-bit data words (WLS = 3).

Table 11-3. UART Parity

PEN	STP	EPS	Data (hex)	Data (binary, LSB first)	Parity
0	x	x	x	x	None
1	0	0	0x60	0000 0110	1
1	0	0	0x57	1110 1010	0
1	0	1	0x60	0000 0110	0
1	0	1	0x57	1110 1010	1
1	1	0	x	x	1
1	1	1	x	x	0

If set, the SB bit forces the TX pin to low asynchronously, regardless of whether or not data is currently transmitted. It functions even when the

UART clock is disabled. Since the TX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

The DLAB bit controls whether the UART_RBR, UART_THR and UART_IER registers are accessible by the peripheral bus (DLAB = 0) or the divisor latch registers UART_DLH and UART_DLL alternatively (DLAB = 1).

UART Modem Control (UART_MCR) Register

The UART_MCR register controls the UART port, as shown in Figure 11-8. Even if modem functionality is not supported, the UART_MCR register is available in order to support the loopback mode.

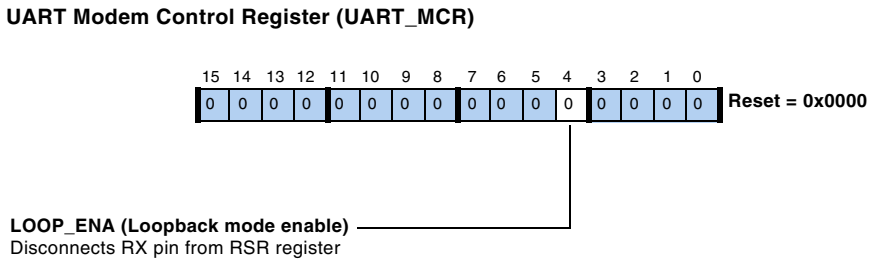

UART Modem Control Register (UART_MCR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
LOOP_ENA (Loopback mode enable)
Disconnects RX pin from RSR register

Figure 11-8. UART Modem Control Registers

Loopback mode disconnects the receiver's input from the RX pin, but redirects it to the transmit output internally.

UART Line Status (UART_LSR) Register

The UART_LSR register contains UART status information as shown in Figure 11-9.

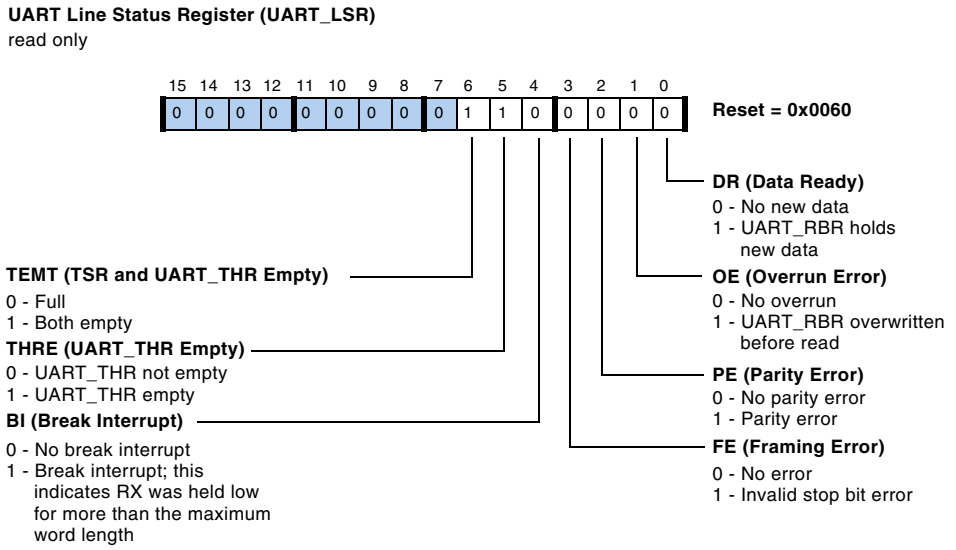

UART Line Status Register (UART_LSR)
read only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
Reset = 0x0060
DR (Data Ready)
0 - No new data
1 - UART_RBR holds new data
OE (Overrun Error)
0 - No overrun
1 - UART_RBR overwritten before read
PE (Parity Error)
0 - No parity error
1 - Parity error
FE (Framing Error)
0 - No error
1 - Invalid stop bit error
TEMT (TSR and UART_THR Empty)
0 - Full
1 - Both empty
THRE (UART_THR Empty)
0 - UART_THR not empty
1 - UART_THR empty
BI (Break Interrupt)
0 - No break interrupt
1 - Break interrupt; this indicates RX was held low for more than the maximum word length

Figure 11-9. UART Line Status Register

The DR bit indicates that data is available in the receiver and can be read from the UART_RBR register. The bit is set by hardware when the receiver detects the first valid stop bit. It is cleared by hardware when the UART_RBR register is read.

The OE bit indicates that a start bit condition has been detected, but the internal receive shift register (RSR) and the receive buffer (UART_RBR) already contain data. New data overwrites the content of the buffers. To avoid overruns read the UART_RBR register in time. The OE bit cleared when the UART_LSR register is read.

The PE bit indicates that the received parity bit does not match the expected value. The PE bit is set simultaneously with the DR bit. The PE bit

cleared when the UART_LSR register is read. Invalid parity bits can be simulated by setting the FPE bit in the UART_GCTL register.

The FE bit indicates that the first stop bit has been sampled low. It is cleared by hardware when the UART_RBR register is read. Invalid stop bits can be simulated by setting the FFE bit in the UART_GCTL register.

The BI bit indicates that the first stop bit has been sampled low and the entire data word, including parity bit, consists of low bits only. It is cleared by hardware when the UART_RBR register is read.

Because of the destructive nature of these read operations, special care should be taken. For more information, see the *Memory* chapter of the *ADSP-BF59x Blackfin Processor Hardware Reference*.

The THRE bit indicates that the UART transmit channel is ready for new data and software can write to UART_THR. Writes to UART_THR clear the THRE bit. It is set again when data is passed from UART_THR to the internal TSR register.

The TEMT bit indicates that both the UART_THR register and the internal TSR register are empty. In this case the program is permitted to write to the UART_THR register twice without losing data. The TEMT bit can also be used as an indicator that pending UART transmission has been completed. At that time it is safe to disable the UCEN bit or to three-state the off-chip line driver.

UART Transmit Holding (UART_THR) Register

The write-only UART_THR register, shown in Figure 11-10, is mapped to the same address as the read-only UART_RBR and UART_DLL registers. To access UART_THR, the DLAB bit in UART_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UART_THR register, and reads from this address return the UART_RBR register.

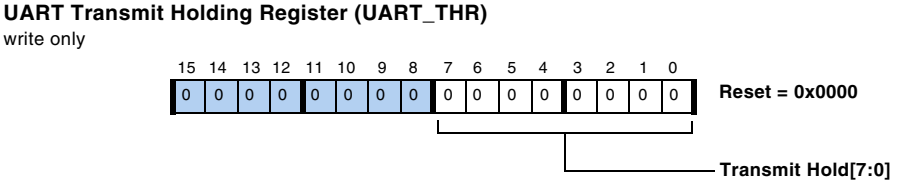

UART Transmit Holding Register (UART_THR)
write only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Transmit Hold[7:0]

Figure 11-10. UART Transmit Holding Register

UART Receive Buffer (UART_RBR) Register

The read-only UART_RBR register, shown in Figure 11-11, is mapped to the same address as the write-only UART_THR and UART_DLL registers. To access UART_RBR, the DLAB bit in UART_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UART_THR register, while reads from this address return the value in the UART_RBR register.

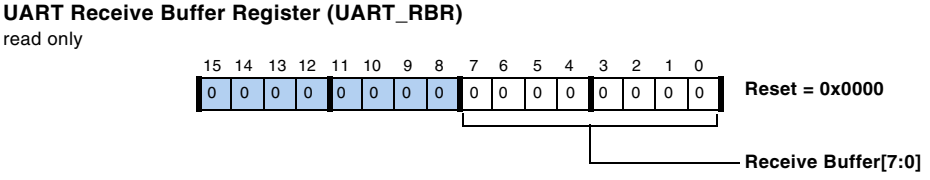

UART Receive Buffer Register (UART_RBR)
read only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Receive Buffer[7:0]

Figure 11-11. UART Receive Buffer Register

UART Interrupt Enable (UART_IER) Register

The UART_IER register, shown in Figure 11-12 on page 11-27, is used to enable requests for system handling of empty or full states of UART data registers. Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and

appropriate interrupt handling routines must be present. For backward compatibility, the UART_IIR still reflects the correct interrupt status.

Each UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless of whether DMA is enabled or not. On some processors, the status interrupt channels from multiple UARTs may be ORed prior to being connected to the system interrupt controller. See Chapter 4, "System Interrupts" for more information.

With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

The UART_IER registers are mapped to the same address as the UART_DLH registers. To access UART_IER, the DLAB bit in UART_LCR must be cleared.

UART Interrupt Enable Register (UART_IER)

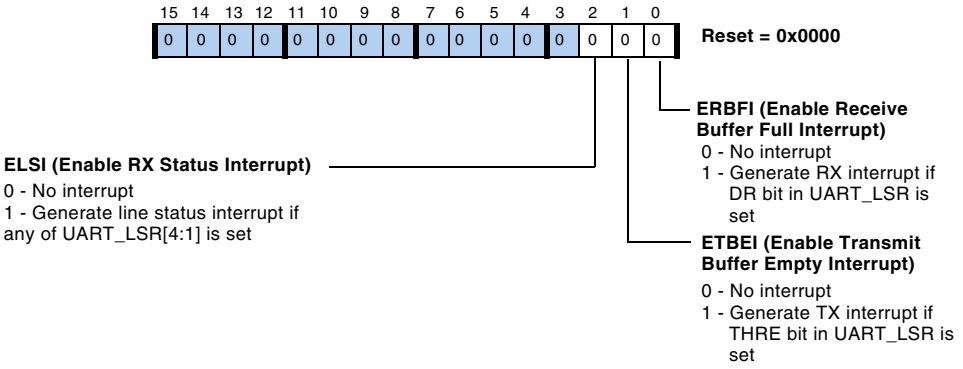

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ERBFI (Enable Receive Buffer Full Interrupt)
0 - No interrupt
1 - Generate RX interrupt if DR bit in UART_LSR is set
ETBEI (Enable Transmit Buffer Empty Interrupt)
0 - No interrupt
1 - Generate TX interrupt if THRE bit in UART_LSR is set
ELSI (Enable RX Status Interrupt)
0 - No interrupt
1 - Generate line status interrupt if any of UART_LSR[4:1] is set

Figure 11-12. UART Interrupt Enable Register

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UART_IER register. This is because the interrupt request lines double as

DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The ELSI bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the UART_LSR register:

- Receive overrun error (OE)
- Receive parity error (PE)
- Receive framing error (FE)
- Break interrupt (BI)

UART Interrupt Identification (UART_IIR) Register

The UART_IIR register conveys interrupt status within the UART. When cleared, the NINT bit signals that an interrupt is pending. The STATUS field indicates the highest priority pending interrupt. The receive line status has the highest priority; the UART_THR empty interrupt has the lowest priority. In the case where both interrupts are signaling, the UART_IIR reads 0x06.

When a UART interrupt is pending, the interrupt service routine needs to clear the interrupt latch explicitly. Figure 11-13 shows how to clear any of the three latches.

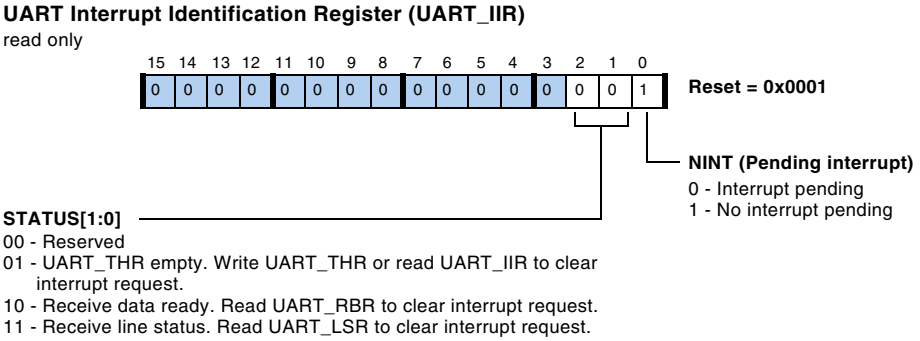

UART Interrupt Identification Register (UART_IIR)
read only
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Reset = 0x0001
NINT (Pending interrupt)
0 - Interrupt pending
1 - No interrupt pending
STATUS[1:0]
00 - Reserved
01 - UART_THR empty. Write UART_THR or read UART_IIR to clear interrupt request.
10 - Receive data ready. Read UART_RBR to clear interrupt request.
11 - Receive line status. Read UART_LSR to clear interrupt request.

Figure 11-13. UART Interrupt Identification Register

The TX interrupt request is cleared by writing new data to the UART_THR register or by reading the UART_IIR register. Please note the special role of the UART_IIR register read in the case where the service routine does not want to transmit further data.

If software stops transmission, it must read the UART_IIR register to reset the interrupt request. As long as the UART_IIR register reads 0x04 or 0x06 (indicating that another interrupt of higher priority is pending), the UART_THR empty latch cannot be cleared by reading UART_IIR.

Because of the destructive nature of these read operations, special care should be taken. For more information, see the *Memory* chapter of the *ADSP-BF59x Blackfin Processor Hardware Reference*.

UART Divisor Latch (UART_DLL and UART_DLH) Registers

The UART_DLL register is mapped to the same address as the UART_THR and UART_RBR registers. The UART_DLH register is mapped to the same address as the UART_IER register. The DLAB bit in UART_LCR must be set before the UART_DLL and UART_DLH registers, shown in Figure 11-14, can be accessed.

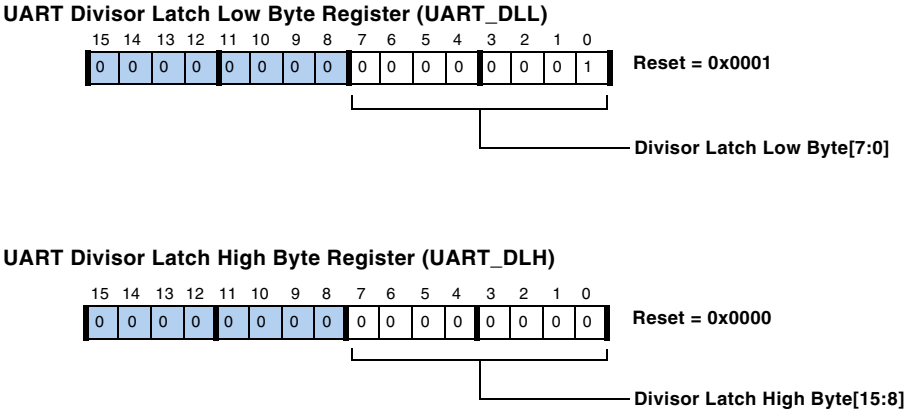

UART Divisor Latch Low Byte Register (UART_DLL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Reset = 0x0001
Divisor Latch Low Byte[7:0]
UART Divisor Latch High Byte Register (UART_DLH)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Divisor Latch High Byte[15:8]

Figure 11-14. UART Divisor Latch Registers

Note the 16-bit divisor formed by UART_DLH and UART_DLL resets to 0x0001, resulting in the highest possible clock frequency by default. If the UART is not used, disabling the UART clock saves power. The UART_DLH and UART_DLL registers can be programmed by software before or after setting the UCEN bit.

UART Scratch (UART_SCR) Register

The 8-bit UART_SCR register, shown in Figure 11-15, is used for general-purpose data storage and does not control the UART hardware in any way. The contents are reset to 0x00.

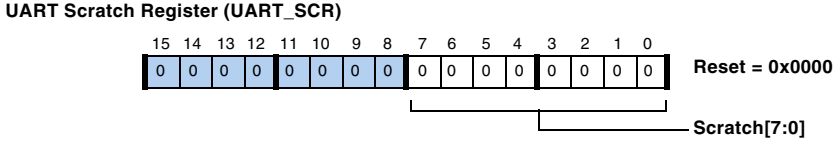


Figure 11-15. UART Scratch Register

UART Global Control (UART_GCTL) Register

The UART_GCTL register, shown in Figure 11-16, contains the enable bit for internal UART clocks and for the IrDA mode of operation of the UART.

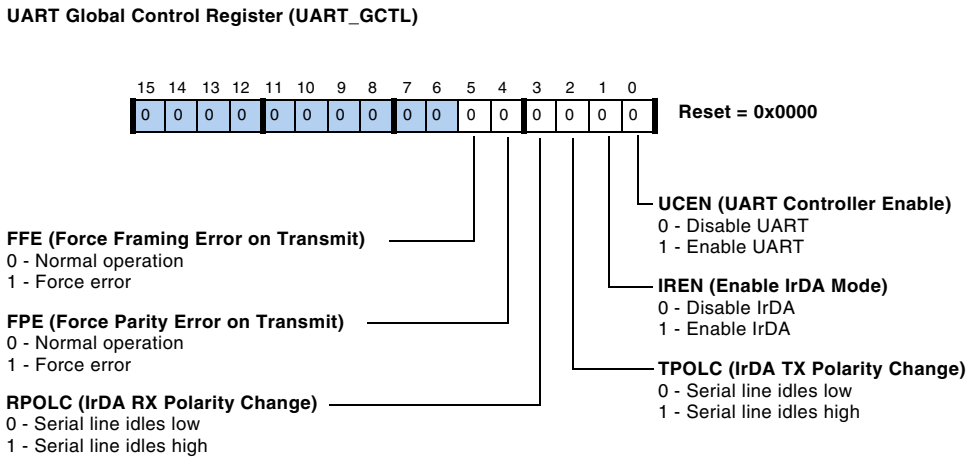

UART Global Control Register (UART_GCTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
FFE (Force Framing Error on Transmit)
0 - Normal operation
1 - Force error
FPE (Force Parity Error on Transmit)
0 - Normal operation
1 - Force error
RPOLC (IrDA RX Polarity Change)
0 - Serial line idles low
1 - Serial line idles high
UCEN (UART Controller Enable)
0 - Disable UART
1 - Enable UART
IREN (Enable IrDA Mode)
0 - Disable IrDA
1 - Enable IrDA
TPOLC (IrDA TX Polarity Change)
0 - Serial line idles low
1 - Serial line idles high

Figure 11-16. UART Global Control Register

The UCEN bit enables the UART clocks. It also resets the state machine and control registers when cleared.

This bit has been introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX polarity change bit and the IrDA RX polarity change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loopback mode.

Programming Examples

The subroutine in Listing 11-1 shows a typical UART initialization sequence.

Listing 11-1. UART Initialization

```
/*************************************************************
 * Configures UART in 8 data bits, no parity, 1 stop bit mode.
 * Input parameters: r0 holds divisor latch value to be
 *                   written into
 *                   DLH:DLL registers.
 *                   p0 contains the UART_GCTL register address
 *  Return values:    none
 *************************************************************/
uart_init:
   [--sp] = r7;
   r7 = UCEN (z);  /* First of all, enable UART clock */
   w[p0+UART0_GCTL-UART0_GCTL] = r7;
   r7 = DLAB (z);                      /* to set bit rate    */
   w[p0+UART0_LCR-UART0_GCTL] = r7;    /* set DLAB bit first */
   w[p0+UART0_DLL-UART0_GCTL] = r0; /* write lower byte to DLL */
   r7 = r0 >> 8;
   w[p0+UART0_DLH-UART0_GCTL] = r7; /* write upper byte to DLH */
   r7 = STB | WLS(8) (z);     /* clear DLAB again and config to */
   w[p0+UART0_LCR-UART0_GCTL] = r7;
                            /* 8 bits, no parity, 2 stop bits */
   r7 = [sp++];
   rts;
uart_init.end:
```

The subroutine in Listing 11-2 performs autobaud detection similarly to UART boot.

Listing 11-2. UART Autobaud Detection Subroutine

```
/*************************************************************
 *  Assuming 8 data bits, this functions expects a '@'
 *  (ASCII 0x40) character
 *  on the UART RX pin. A Timer performs the autobaud detection.
 *  Input parameters: p0 contains the UART_GCTL register address
 *                    p1 contains the TIMER_CONFIG register
 *  address
 *  Return values:    r0 holds timer period value (equals 8 bits)
*************************************************************/
uart_autobaud:
   [--sp] = (r7:5,p5:5);
   r5.h = hi(TIMER0_CONFIG); /* for generic timer use calculate
*/
   r5.l = lo(TIMER0_CONFIG);   /* specific bits first */
   r7 = p1;
   r7 = r7 - r5;
   r7 >>= 4;         /* r7 holds the 'x' of TIMERx_CONFIG now */
   r5 = TIMEN0 (z);
   r5 <<= r7;        /* r5 holds TIMENx/TIMDISx now */
   r6 = TRUN0 | TOVL_ERR0 | TIMIL0 (z);
   r6 <<= r7;
   CC = r7 <= 3;
   r7 = r6 << 12;
   if !CC r6 = r7;   /* r6 holds TRUNx | TOVL_ERRx | TIMILx */
   p5.h = hi(TIMER_STATUS);
   p5.l = lo(TIMER_STATUS);
   w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
   [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
               /* clear pending latches */
               /* period capture, falling edge to falling edge */
   r7 = TIN_SEL | IRQ_ENA | PERIOD_CNT | WDTH_CAP (z);
```

```
   w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
   w[p5+TIMER_ENABLE-TIMER_STATUS] = r5;
uart_autobaud.wait:   /* wait for timer event */
         r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
         r7 = r7 & r5;
         CC = r7 == 0;
         if CC jump uart_autobaud.wait;
   w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
   [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
                              /* clear pending latches */
   /* Save period value to R0 */
   r0 = [p1 + TIMER0_PERIOD - TIMER0_CONFIG];
   /* delay processing as autobaud character is still ongoing */
   r7 = OUT_DIS | IRQ_ENA | PERIOD_CNT | PWM_OUT (z);
   w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
   w[p5 + TIMER_ENABLE - TIMER_STATUS] = r5;
uart_autobaud.delay:
         r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
         r7 = r7 & r5;
         CC = r7 == 0;
         if CC jump uart_autobaud.delay;
   w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5;
   [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
   (r7:5,p5:5) = [sp++];
   rts;
uart_autobaud.end:
```

The parent routine in Listing 11-3 performs autobaud detection, using as an example a processor whose TIMER4 is mapped to UART0 for this purpose. Note also that this example assumes the processor's UART0 pins are mapped to PORT G (PG7 and PG8).

Listing 11-3. UART Autobaud Detection Parent Routine

```
p0.l = lo(PORTG_FER);
         /* function enable on UART0 pins PG7 and PG8 */
p0.h = hi(PORTG_FER);
r0 = PG7 | PG8 (z)
w[p0] = r0;
p0.l = lo(PORTG_MUX);
p0.h = hi(PORTG_MUX);
r0.l = 0x0020;
r0.h = 0x0000;
w[p0] = r0;
p0.l = lo(UART0_GCTL);      /* select UART 0 */
p0.h = hi(UART0_GCTL);
p1.l = lo(TIMER4_CONFIG);   /* select TIMER 4 */
p1.h = hi(TIMER4_CONFIG);
call uart_autobaud;
r0 >>= 7;      /* divide PERIOD value by (16 x 8) */
call uart_init;
...
```

The subroutine in Listing 11-4 on page 11-35 transmits a character by polling operation.

Listing 11-4. UART Character Transmission

```
/*******************************************************
 *  Transmit a single byte by polling the THRE bit.
 *  Input parameters: r0 holds the character to be transmitted
 *                    p0 contains UART_GCTL register address
 *  Return values: none
*******************************************************/
uart_putc:
   [--sp] = r7;
uart_putc.wait:
```

```
         r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
         CC = bittst(r7, bitpos(THRE));
         if !CC jump uart_putc.wait;
   w[p0+UART0_THR-UART0_GCTL] = r0; /* write initiates transfer
*/
   r7 = [sp++];
   rts;
uart_putc.end:
```

Use the routine shown in Listing 11-5 to transmit a C-style string that is terminated by a null character.

Listing 11-5. UART String Transmission

```
/*************************************************************
 *  Transmit a null-terminated string.
 *  Input parameters: p1 points to the string
 *                    p0 contains UART_GCTL register address
 *  Return values: none
*************************************************************/
uart_puts:
   [--sp] = rets;
   [--sp] = r0;
uart_puts.loop:
         r0 = b[p1++] (z);
         CC = r0 == 0;
         if CC jump uart_puts.exit;
         call uart_putc;
         jump uart_puts.loop;
uart_puts.exit:
   r0 = [sp++];
   rets = [sp++];
   rts;
uart_puts.end:
```

Note that polling the UART0_LSR register for transmit purposes may clear the receive error latch bits. It is, therefore, not recommended to poll UART0_LSR for transmission this way while data is being received. In that case, write a polling loop that reads UART_LSR once and then evaluates *all* status bits of interest, as shown in Listing 11-6.

Listing 11-6. UART Polling Loop

```
uart_loop:
   r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
   CC = bittst(r7, bitpos(DR));
   if !CC jump uart_loop.transmit;
   r6 = w[p0+UART0_RBR-UART0_GCTL] (z);
   r5 = BI | OE | FE | PE (z);
   r5 = r5 & r7;
   CC = r5 == 0;
   if !CC jump uart_loop.error;
   b[p1++] = r6;          /* store byte */
uart_loop.transmit:
   CC = bittst(r7, bitpos(THRE));
   if !CC jump uart_loop;
   r5 = b[p2++] (z);      /* load next byte */
   w[p0+UART0_THR-UART0_GCTL] = r5;
   jump uart_loop;
uart_loop.error:
   ...
   jump uart_loop;
```

In non-DMA interrupt operation, the three UART interrupt request lines may or may not be ORed together in the system interrupt controller. If they had three different service routines, they may look as shown in Listing 11-7.

Listing 11-7. UART Non-DMA Interrupt Operation

```
isr_uart_rx:
   [--sp] = astat;
   [--sp] = r7;
   r7 = w[p0+UART0_RBR-UART0_GCTL] (z);
   b[p4++] = r7;
   ssync;
   r7 = [sp++];
   astat = [sp++];
   rti;
isr_uart_rx.end:
isr_uart_tx:
   [--sp] = astat;
   [--sp] = r7;
   r7 = b[p3++] (z);
   CC = r7 == 0;
   if CC jump isr_uart_tx.final;
   w[p0+UART0_THR-UART0_GCTL] = r7;
   r7 = [sp++];
   astat = [sp++];
   ssync;
   rti;
isr_uart_tx.final:
   r7 = w[p0+UART0_IER-UART0_GCTL] (z);
               /* clear TX interrupt enable */
   bitclr(r7, bitpos(ETBEI));   /* ensure this sequence is not */
   w[p0+UART0_IER-UART0_GCTL] = r7;
               /* interrupted by other IER accesses */
   ssync;
   r7 = [sp++];
   astat = [sp++];
   rti;
isr_uart_tx.end:
```

```
isr_uart_error:
   [--sp] = astat;
   [--sp] = r7;
   r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
              /* read clears interrupt request */
              /* do something with the error */
   r7 = [sp++];
   astat = [sp++];
   ssync;
   rti;
isr_uart_error.end:
```

Listing 11-8 transmits a string by DMA operation, waits until DMA completes and sends an additional string by polling. Note the importance of the SYNC bit.

Listing 11-8. UART Transmission SYNC Bit Use

```
.section data;
.byte sHello[] = 'Hello Blackfin User',13,10,0;
.byte sWorld[] = 'How is life?',13,10,0;
.section program;
   ...
   p1.l = lo(IMASK);
   p1.h = hi(IMASK);
   r0.l = lo(isr_uart_tx);      /* register service routine */
   r0.h = hi(isr_uart_tx); /*Assume UART0 TX defaults to IVG10*/
   r0 = [p1 + IMASK - IMASK];   /* unmask interrupt in CEC */
   bitset(r0, bitpos(EVT_IVG10));
   [p1] = r0;
   p1.l = lo(SIC_IMASK0);
   p1.h = hi(SIC_IMASK0);
/* unmask interrupt in SIC */
/* (assume SIC_IMASK0 for this example)*/
   r0.l = 0x0080;
```

```
    r0.h = 0x0000;
    [p1] = r0;
    [--sp] = reti;    /* enable nesting of interrupts */
    p5.l = lo(DMA9_CONFIG);
/* setup DMA in STOP mode */
/* (assume DMA channel 9 for this example)*/
    p5.h = hi(DMA9_CONFIG);
    r7.l = lo(sHello);
    r7.h = hi(sHello);
    [p5+DMA9_START_ADDR-DMA9_CONFIG] = r7;
    r7 = length(sHello) (z);
    r7+= -1;           /* do not send trailing null character */
    w[p5+DMA9_X_COUNT-DMA9_CONFIG] = r7;
    r7 = 1;
    w[p5+DMA9_X_MODIFY-DMA9_CONFIG] = r7;
    r7 = FLOW_STOP | WDSIZE_8 | DI_EN | SYNC | DMAEN (z);
    w[p5] = r7;
    p0.l = lo(UART0_GCTL);   /* select UART 0 */
    p0.h = hi(UART0_GCTL);
    r0 = ETBEI (z);          /* enable and issue first request */
    w[p0+UART0_IER-UART0_GCTL] = r0;
wait4dma:  /* just one way to synchronize with the service rou-
tine */
    r0 = w[p5+DMA9_IRQ_STATUS-DMA9_CONFIG] (z);
    CC = bittst(r0,bitpos(DMA_RUN));
    if CC jump wait4dma;
    p1.l=lo(sWorld);
    p1.h=hi(sWorld);
    call uart_puts;
forever: jump forever;
isr_uart_tx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = DMA_DONE (z);   /* W1C interrupt request */
```

```
    w[p5+DMA9_IRQ_STATUS-DMA9_CONFIG] = r7;
    r7 = 0;              /* pulse ETBEI for general case */
    w[p0+UART0_IER-UART0_GCTL] = r7;
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:
```

Unique Information for the ADSP-BF59x Processor

None.

12 TWO WIRE INTERFACE CONTROLLER

This chapter describes the two wire interface (TWI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of TWIs for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For TWI interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

To determine how each of the TWIs is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each TWI, refer to Chapter A, "System MMR Assignments".

TWI behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 12-56.

Overview

The TWI controller allows a device to interface to an inter IC bus as specified by the *Philips I²C Bus Specification version 2.1* dated January 2000.

The TWI is fully compatible with the widely used I²C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth the TWI controller can be set up with transfer initiated interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

The TWI externally moves 8-bit data while maintaining compliance with the I²C bus protocol. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master bus arbitration
- 7-bit addressing
- 100K bits/second and 400K bits/second data rates
- General call address support
- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up

- Input filter for spike suppression
- Serial camera control bus support as specified in the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification* version 2.1.

Interface Overview

Figure 12-1 provides a block diagram of the TWI controller. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCL rate, to and from other TWI devices. The SCL signal synchronizes the shifting and sampling of the data on the serial data pin.

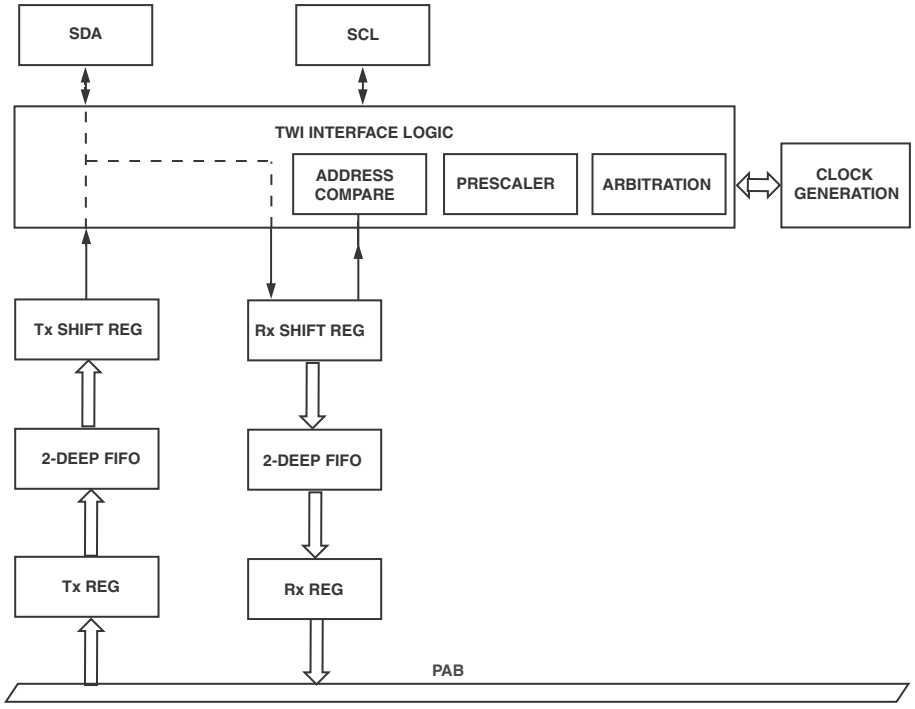


Figure 12-1. TWI Block Diagram

External Interface

The SDA (serial data) and SCL (serial clock) signals are open drain and as such require pull-up resistors.

Serial Clock Signal (SCL)

In slave mode this signal is an input and an external master is responsible for providing the clock.

In master mode the TWI controller must set this signal to the desired frequency. The TWI controller supports the standard mode of operation (up to 100 KHz) or fast mode (up to 400 KHz).

The TWI control register (TWI_CONTROL) is used to set the PRESCALE value which gives the relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

$$\text{PRESCALE} = f_{SCLK}/10\text{MHz}$$

The PRESCALE value is the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

Serial Data Signal (SDA)

This is a bidirectional signal on which serial data is transmitted or received depending on the direction of the transfer.

TWI Pins

Table 12-1 shows the pins for the TWI. Two bidirectional pins externally interface the TWI controller to the I^2C bus. The interface is simple and no other external connections or logic are required.

Table 12-1. TWI Pins

Pin	Description
SDA	In/Out TWI serial data, high impedance reset value.
SCL	In/Out TWI serial clock, high impedance reset value.

Internal Interfaces

The peripheral bus interface supports the transfer of 16-bit wide data and is used by the processor in the support of register and FIFO buffer reads and writes.

The register block contains all control and status bits and reflects what can be written or read as outlined by the programmer's model. Status bits can be updated by their respective functional blocks.

The FIFO buffer is configured as a1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event the TWI controller module is accessed as a slave.

The prescaler block must be programmed to generate a 10 MHz time reference relative to the system clock. This time base is used for filtering of data and timing events specified by the electrical data sheet (See the Philips Specification), as well as for SCL clock generation.

The clock generation module is used to generate an external SCL clock when in master mode. It includes the logic necessary for synchronization in a multi-master clock configuration and clock stretching when configured in slave mode.

Description of Operation

The following sections describe the operation of the TWI interface.

TWI Transfer Protocols

The TWI controller follows the transfer protocol of the *Philips I^2C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in Figure 12-2.

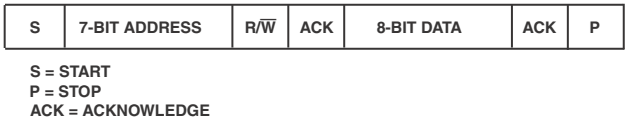


Figure 12-2. Basic Data Transfer

To better understand the mapping of TWI controller register contents to a basic transfer, Figure 12-3 details the same transfer as above noting the corresponding TWI controller bit names. In this illustration, the TWI controller successfully transmits one byte of data. The slave has acknowledged both address and data.

Figure 12-3. Data Transfer With Bit Illustration

Clock Generation and Synchronization

The TWI controller implementation only issues a clock during master mode operation and only at the time a transfer has been initiated. If arbitration for the bus is lost, the serial clock output immediately three-states. If multiple clocks attempt to drive the serial clock line, the TWI controller synchronizes its clock with the other remaining clocks. This is shown in Figure 12-4.

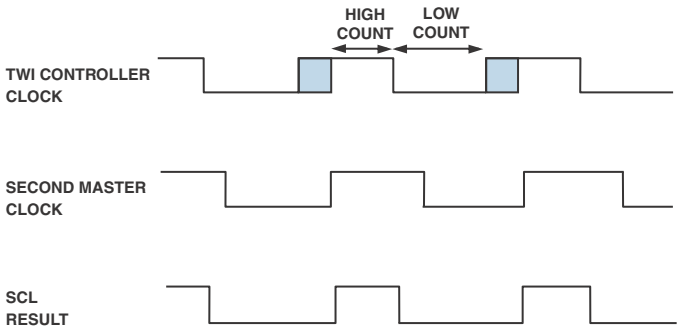


Figure 12-4. TWI Clock Synchronization

The TWI controller's serial clock (SCL) output follows these rules:

- Once the clock high (CLKHI) count is complete, the serial clock output is driven low and the clock low (CLKLOW) count begins.
- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCL line is detected at a logic 1 level. At this time the clock high count begins.

Bus Arbitration

The TWI controller initiates a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 12-5.

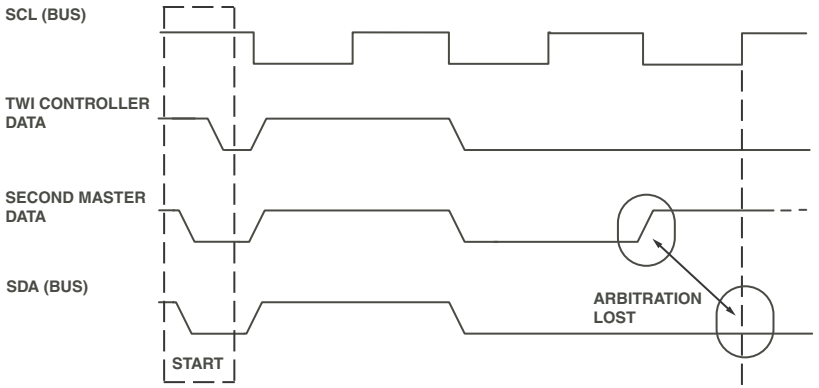


Figure 12-5. TWI Bus Arbitration

The TWI controller monitors the serial data bus (SDA) while SCL is high and if SDA is determined to be an active logic 0 level while the TWI controller's data is a logic 1 level, the TWI controller has lost arbitration and ends generation of clock and data. Note arbitration is not performed only at serial clock edges, but also during the entire time SCL is high.

Start and Stop Conditions

Start and stop conditions involve serial data transitions while the serial clock is a logic 1 level. The TWI controller generates and recognizes these transitions. Typically start and stop conditions occur at the beginning and at the conclusion of a transmission with the exception repeated start "combined" transfers, as shown in Figure 12-6.

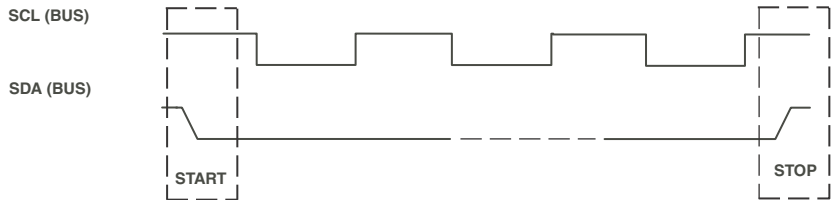


Figure 12-6. TWI Start and Stop Conditions

The TWI controller's special case start and stop conditions include:

- TWI controller addressed as a slave-receiver

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP).

- TWI controller addressed as a slave-transmitter

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP) and indicates a slave transfer error (SERR).

- TWI controller as a master-transmitter or master-receiver

  If the stop bit is set during an active master transfer, the TWI controller issues a stop condition as soon as possible avoiding any error conditions (as if data transfer count had been reached).

General Call Support

The TWI controller always decodes and acknowledges a general call address if it is enabled as a slave (SEN) and if general call is enabled (GEN). general call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK'ed, the NAK bit can be set.

If the TWI controller is to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most affected. When fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation (t_r)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a stop and start condition (t_{BUF})

Functional Description

The following sections describe the functional operation of the TWI.

General Setup

General setup refers to register writes that are required for both slave mode operation and master mode operation. General setup should be performed before either the master or slave enable bits are set.

- Program the TWI_CONTROL register to enable the TWI controller and set the prescale value. Program the prescale value to the binary representation of f_{SCLK}/10MHz

All values should be rounded up to the next whole number. The TWI_ENA bit enable must be set. Note once the TWI controller is enabled a bus

busy condition may be detected. This condition should clear after t_{BUF} has expired assuming no additional bus activity has been detected.

Slave Mode

When enabled, slave mode operation supports both receive and transmit data transfers. It is not possible to enable only one data transfer direction and not acknowledge (NAK) the other. This is reflected in the following setup.

1. Program TWI_SLAVE_ADDR. The appropriate 7 bits are used in determining a match during the address phase of the transfer.
2. Program TWI_XMT_DATA8 or TWI_XMT_DATA16. These are the initial data values to be transmitted in the event the slave is addressed and a transmit is required. This is an optional step. If no data is written and the slave is addressed and a transmit is required, the serial clock (SCL) is stretched and an interrupt is generated until data is written to the transmit FIFO.
3. Program TWI_INT_MASK. Enable bits are associated with the desired interrupt sources. As an example, programming the value 0x000F results in an interrupt output to the processor in the event that a valid address match is detected, a valid slave transfer completes, a slave transfer has an error, a subsequent transfer has begun yet the previous transfer has not been serviced.
4. Program TWI_SLAVE_CTL. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation, requires 7-bit addressing, and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 12-2 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 12-2. Slave Mode Setup Interaction

TWI Controller Master	Processor
Interrupt: SINIT – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: SCOMP – Slave transfer complete.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.

Master Mode Clock Setup

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

- Program `TWI_CLKDIV`. This defines the clock high duration and clock low duration.

Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program `TWI_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWI_XMT_DATA8` or `TWI_XMT_DATA16`. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.

3. Program TWI_FIFO_CTL. Indicate if transmit FIFO buffer interrupts should occur with each byte transmitted (8-bits) or with each two bytes transmitted (16-bits).

4. Program TWI_INT_MASK. Enable bits associated with the desired interrupt sources. As an example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.

5. Program TWI_MASTER_CTL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, generates a 7-bit address, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a Stop condition.

Table 12-3 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 12-3. Master Mode Transmit Setup Interaction

TWI Controller Master	Processor
Interrupt: XMTEMPTY – Transmit buffer is empty.	Write transmit FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits.

Master Mode Receive

Follow these programming steps for a single master mode receive:

1. Program TWI_MASTER_ADDR. This defines the address transmitted during the address phase of the transfer.
2. Program TWI_FIFO_CTL. Indicate if receive FIFO buffer interrupts should occur with each byte received (8-bits) or with each two bytes received (16-bits).
3. Program TWI_INT_MASK. Enable bits associated with the desired interrupt sources. For example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.
4. Program TWI_MASTER_CTL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, generates a 7-bit address, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a Stop condition.

Table 12-4 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 12-4. Master Mode Receive Setup Interaction

TWI Controller Master	Processor
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.

Repeated Start Condition

In general, a repeated start condition is the absence of a stop condition between two transfers. The two transfers can be of any direction type. Examples include a transmit followed by a receive, or a receive followed by a transmit. The following sections guide the programmer in developing a service routine.

Transmit/Receive Repeated Start Sequence

Figure 12-7 shows a repeated start data transmit followed by a data receive sequence.

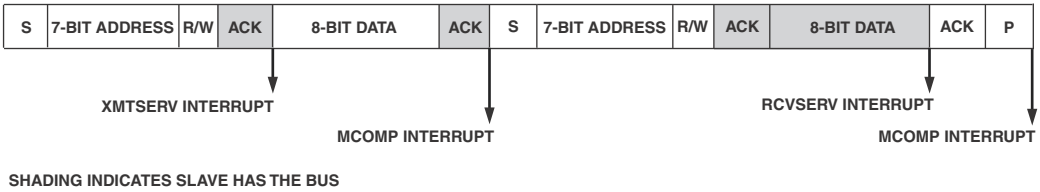


Figure 12-7. Transmit/Receive Data Repeated Start

The following tasks are performed at each interrupt.

- `XMTSERV` interrupt

  This interrupt was generated due to a FIFO access. Since this is the last byte of this transfer, `FIFO_STATUS` indicates the transmit FIFO is empty. When read, `DCNT` would be zero. Set the `RSTART` bit to indicate a repeated start and set the `MDIR` bit if the following transfer will be a data receive.

- `MCOMP` interrupt

This interrupt was generated because all data has been transferred (DCNT = 0). If no errors were generated, a start condition is initiated. Clear the RSTART bit and program the DCNT with the desired number of bytes to receive.

- RCVSERV interrupt

  This interrupt is generated due to the arrival of a byte in the receive FIFO. Simple data handling is all that is required.

- MCOMP interrupt

  The transfer is complete.

Receive/Transmit Repeated Start Sequence

Figure 12-8 on page 12-16 illustrates a repeated start data receive followed by a data transmit sequence.

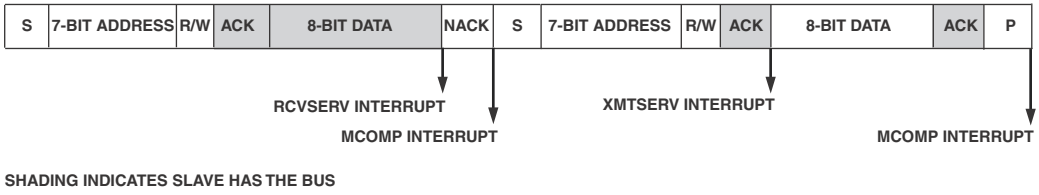


Figure 12-8. Receive/Transmit Data Repeated Start

The tasks performed at each interrupt are:

- RCVSERV interrupt

  This interrupt is generated due to the arrival of a data byte in the receive FIFO. Set the RSTART bit to indicate a repeated start and clear the MDIR bit if the following transfer will be a data transmit.

- MCOMP interrupt

This interrupt has occurred due to the completion of the data receive transfer. If no errors were generated, a start condition is initiated. Clear the RSTART bit and program the DCNT with the desired number of bytes to transmit.

- XMTSERV interrupt

  This interrupt is generated due to a FIFO access. Simple data handling is all that is required.

- MCOMP interrupt

  The transfer is complete.

There is no timing constraint to meet the above conditions—the user can program the bits as required. Refer to "Clock Stretching During Repeated Start Condition" on page 12-20 for more on how the controller stretches the clock during Repeated Start transfers.

Clock Stretching

Clock stretching is an added functionality of the TWI controller in Master Mode operation. This new behavior utilizes self-induced stretching of the I^2C clock while waiting on servicing interrupts. Stretching is done automatically by the hardware and no programming is required for this.

The TWI Controller as Master supports three modes of clock stretching: "Clock Stretching During FIFO Underflow" on page 12-17, "Clock Stretching During FIFO Overflow" on page 12-19 and "Clock Stretching During Repeated Start Condition" on page 12-20.

Clock Stretching During FIFO Underflow

During a master mode transmit, an interrupt is generated at the instant the transmit FIFO becomes empty. At this time, the most recent byte begins transmission. If the XMTSERV interrupt is not serviced, the concluding "acknowledge" phase of the transfer will be stretched. Stretching

of the clock continues until new data bytes are written to the transmit FIFO (TWI_XMT_DATA8 or TWI_XMT_DATA16). No other action is required to release the clock and continue the transmission. This behavior continues until the transmission is complete (DCNT = 0) at which time the transmission is concluded (MCOMP) as shown in Figure 12-9 and described in Table 12-5.

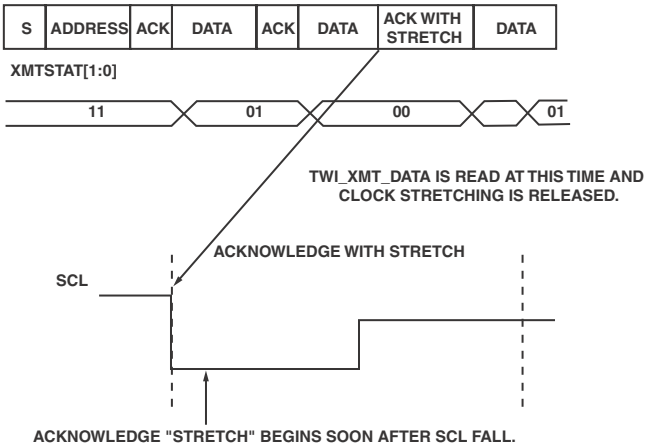


Figure 12-9. Clock Stretching during FIFO Underflow

Table 12-5. FIFO Underflow Case

TWI Controller	Processor
Interrupt: XMTSERV – Transmit FIFO buffer is empty.	Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.
...	...
Interrupt: MCOMP – Master transmit complete (DCNT= 0x00).	Acknowledge: Clear interrupt source bits.

Clock Stretching During FIFO Overflow

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (TWI_RCV_DATA8, TWI_RCV_DATA16). No other action is required to release the clock and continue the reception of data. This behavior continues until the reception is complete (DCNT = 0x00) at which time the reception is concluded (MCOMP) as shown in Figure 12-10 and described in Table 12-6.

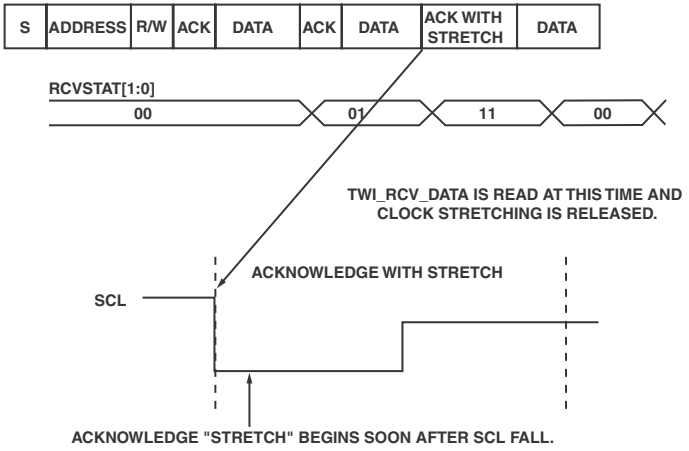


Figure 12-10. Clock Stretching During FIFO Overflow

Table 12-6. FIFO Overflow Case

TWI Controller	Processor
Interrupt: RCVSERV – Receive FIFO buffer is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Clock Stretching During Repeated Start Condition

The repeated start feature in I^2C protocol requires transitioning between two subsequent transfers. With the use of clock stretching, the task of managing transitions becomes simpler and becomes common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock will initiate a stretch during the repeated start phase between transfers. Concurrent with this event the initial transfer will generate a transfer complete interrupt (MCOMP) to signify the initial transfer has completed (DCNT = 0). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in Figure 12-11 and described in Table 12-7.

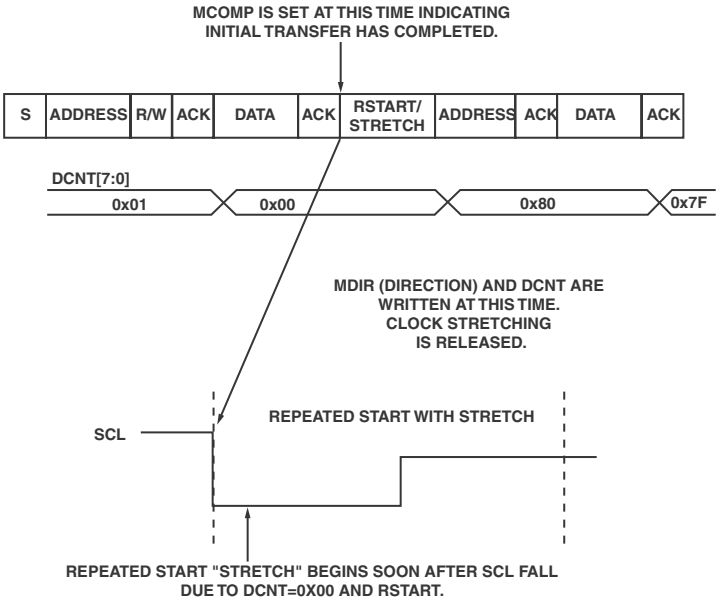


Figure 12-11. Clock Stretching during Repeated Start Condition

Table 12-7. Repeated Start Case

TWI Controller	Processor
Interrupt: MCOMP – Initial transmit has completed and DCNT = 0x00. Note: transfer in progress, RSTART previously set.	Acknowledge: Clear interrupt source bits. Write TWI_MASTER_CTL, setting MDIR (receive), clearing RSTART, and setting new DCNT value (nonzero).
Interrupt: RCVSERV – Receive FIFO is full.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.
...	...
Interrupt: MCOMP – Master receive complete.	Acknowledge: Clear interrupt source bits.

Programming Model

Figure 12-12 and Figure 12-13 illustrate the programming model for the TWI.

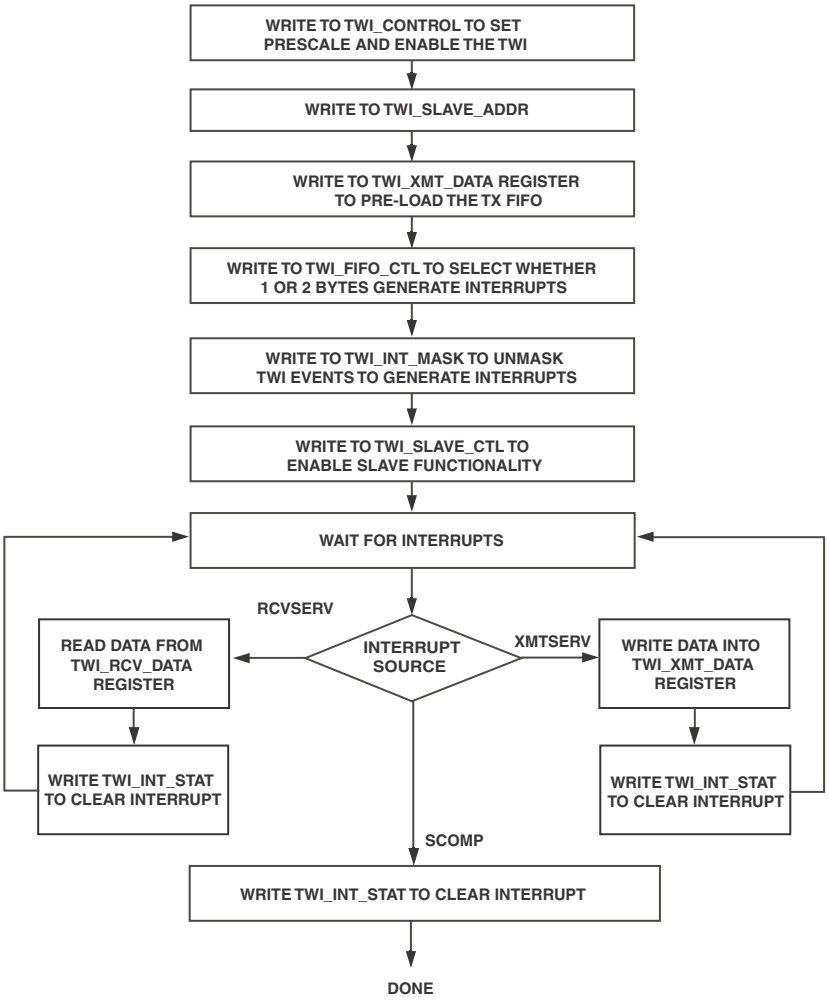

WRITE TO TWI_CONTROL TO SET PRESCALE AND ENABLE THE TWI
WRITE TO TWI_SLAVE_ADDR
WRITE TO TWI_XMT_DATA REGISTER TO PRE-LOAD THE TX FIFO
WRITE TO TWI_FIFO_CTL TO SELECT WHETHER 1 OR 2 BYTES GENERATE INTERRUPTS
WRITE TO TWI_INT_MASK TO UNMASK TWI EVENTS TO GENERATE INTERRUPTS
WRITE TO TWI_SLAVE_CTL TO ENABLE SLAVE FUNCTIONALITY
WAIT FOR INTERRUPTS
RCVSERV
READ DATA FROM TWI_RCV_DATA REGISTER
INTERRUPT SOURCE
XMTSERV
WRITE DATA INTO TWI_XMT_DATA REGISTER
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
SCOMP
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
DONE

Figure 12-12. TWI Slave Mode

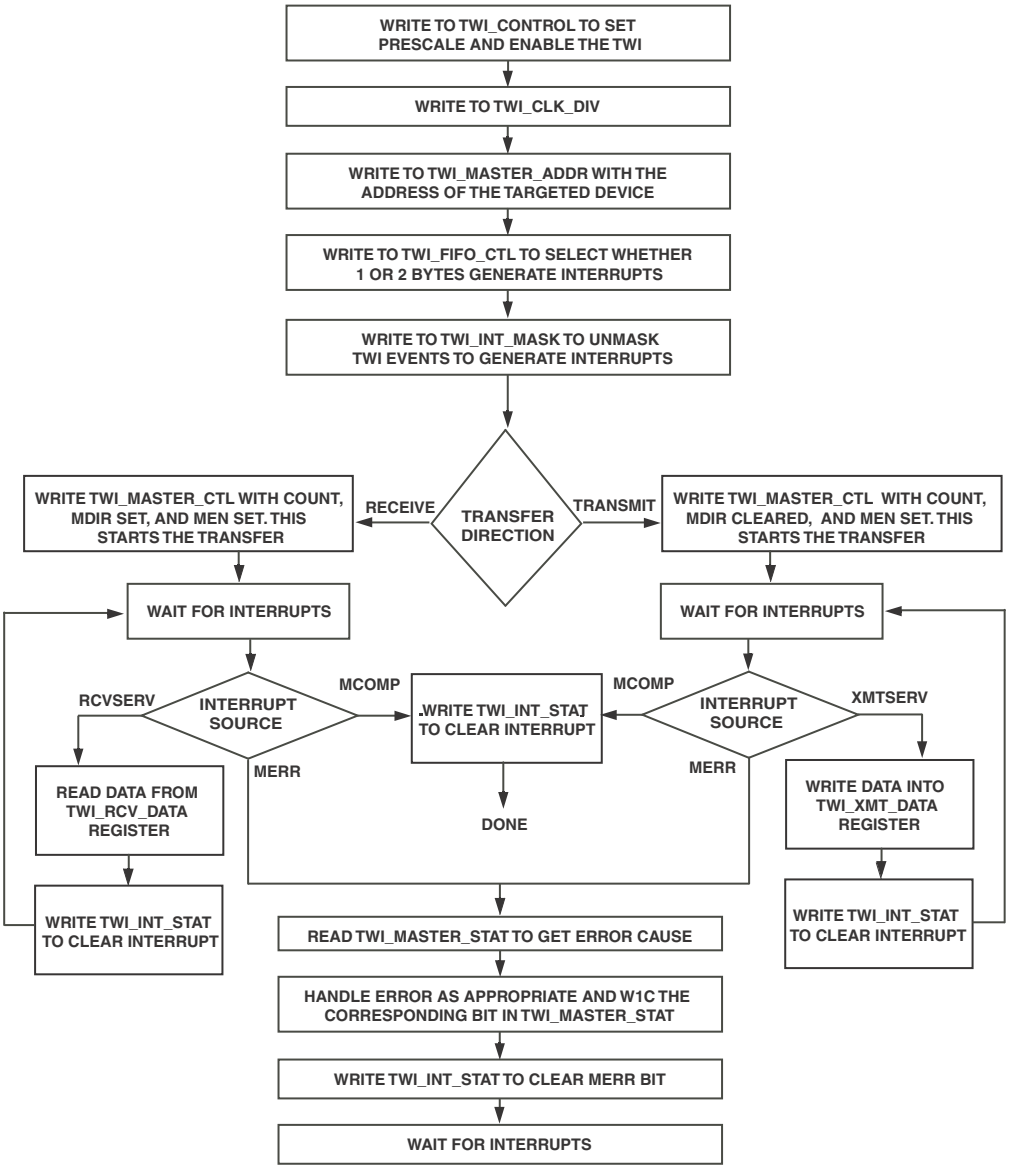

WRITE TO TWI_CONTROL TO SET PRESCALE AND ENABLE THE TWI
WRITE TO TWI_CLK_DIV
WRITE TO TWI_MASTER_ADDR WITH THE ADDRESS OF THE TARGETED DEVICE
WRITE TO TWI_FIFO_CTL TO SELECT WHETHER 1 OR 2 BYTES GENERATE INTERRUPTS
WRITE TO TWI_INT_MASK TO UNMASK TWI EVENTS TO GENERATE INTERRUPTS
TRANSFER DIRECTION
RECEIVE
TRANSMIT
WRITE TWI_MASTER_CTL WITH COUNT, MDIR SET, AND MEN SET. THIS STARTS THE TRANSFER
WRITE TWI_MASTER_CTL WITH COUNT, MDIR CLEARED, AND MEN SET. THIS STARTS THE TRANSFER
WAIT FOR INTERRUPTS
WAIT FOR INTERRUPTS
INTERRUPT SOURCE
RCVSERV
MCOMP
MERR
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
DONE
INTERRUPT SOURCE
MCOMP
MERR
XMTSERV
READ DATA FROM TWI_RCV_DATA REGISTER
WRITE DATA INTO TWI_XMT_DATA REGISTER
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
WRITE TWI_INT_STAT TO CLEAR INTERRUPT
READ TWI_MASTER_STAT TO GET ERROR CAUSE
HANDLE ERROR AS APPROPRIATE AND W1C THE CORRESPONDING BIT IN TWI_MASTER_STAT
WRITE TWI_INT_STAT TO CLEAR MERR BIT
WAIT FOR INTERRUPTS

Figure 12-13. TWI Master Mode

Register Descriptions

The TWI controller has 16 registers described in the following sections. Figure 12-14 through Figure 12-31 on page 12-46 illustrate the registers.

TWI CONTROL Register (TWI_CONTROL)

The TWI_CONTROL register is used to enable the TWI module as well as to establish a relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

PRESCALE = f_{SCLK}/10MHz

SCCB compatibility is an optional feature and should not be used in an I^2C bus system. This feature is turned on by setting the SCCB bit in the TWI_CONTROL register. When this feature is set all slave asserted acknowledgement bits are ignored by this master. This feature is valid only during transfers where the TWI is mastering an SCCB bus. Slave mode transfers should be avoided when this feature is enabled because the TWI controller always generates an acknowledge in slave mode.

For either master and/or slave mode of operation, the TWI controller is enabled by setting the TWI_ENA bit in the TWI_CONTROL register. It is recommended that this bit be set at the time PRESCALE is initialized and remain set. This guarantees accurate operation of bus busy detection logic.

The PRESCALE field of the TWI_CONTROL register specifies the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

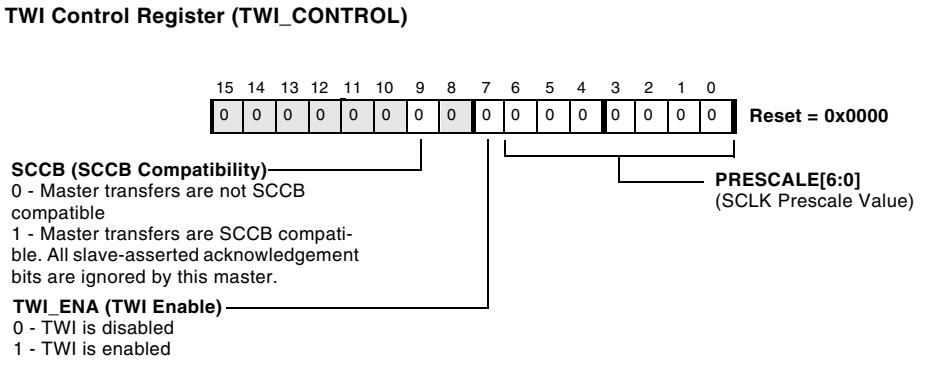

TWI Control Register (TWI_CONTROL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SCCB (SCCB Compatibility)
0 - Master transfers are not SCCB compatible
1 - Master transfers are SCCB compatible. All slave-asserted acknowledgement bits are ignored by this master.
TWI_ENA (TWI Enable)
0 - TWI is disabled
1 - TWI is enabled
PRESCALE[6:0]
(SCLK Prescale Value)

Figure 12-14. TWI Control Register

SCL Clock Divider Register (TWI_CLKDIV)

The clock signal SCL is an output in master mode and an input in slave mode.

During master mode operation, the TWI_CLKDIV register values are used to create the high and low durations of the serial clock (SCL). Serial clock frequencies can vary from 400 KHz to less than 20 KHz. The resolution of the clock generated is 1/10 MHz or 100 ns.

CLKDIV = TWI SCL period / 10 MHz time reference

For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns) and an internal time reference of 10 MHz (period = 100 ns):

CLKDIV = 2500 ns / 100 ns = 25

For an SCL with a 30% duty cycle, then CLKLOW = 17 and CLKHI = 8. Note that CLKLOW and CLKHI add up to CLKDIV.

The CLKHI field of the TWI_CLKDIV register specifies the number of 10 MHz time reference periods the serial clock (SCL) waits before a new clock

low period begins, assuming a single master. It is represented as an 8-bit binary value.

The CLKLOW field of the TWI_CLKDIV register specifies the number of internal time reference periods the serial clock (SCL) is held low. It is represented as an 8-bit binary value.

SCL Clock Divider Register (TWI_CLKDIV)

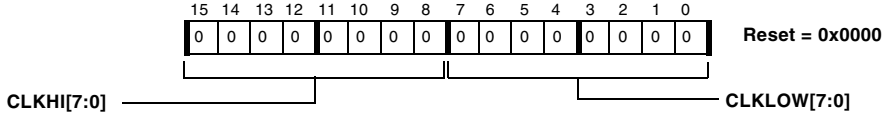


Figure 12-15. SCL Clock Divider Register

TWI Slave Mode Control Register (TWI_SLAVE_CTL)

The TWI_SLAVE_CTL register controls the logic associated with slave mode operation. Settings in this register do not affect master mode operation and should not be modified to control master mode functionality.

TWI Slave Mode Control Register (TWI_SLAVE_CTL)

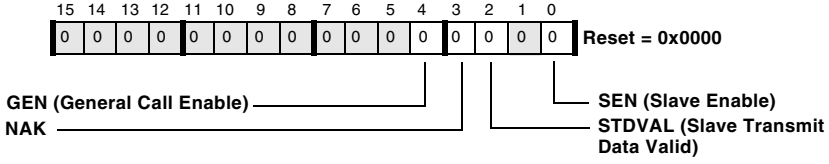

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
GEN (General Call Enable)
NAK
SEN (Slave Enable)
STDVAL (Slave Transmit Data Valid)

Figure 12-16. TWI Slave Mode Control Register

Additional information for the TWI_SLAVE_CTL register bits includes:

- **General call enable** (GEN)

  General call address detection is available only when slave mode is enabled.

  [0] General call address matching is not enabled.

  [1] General call address matching is enabled. A general call slave receive transfer is accepted. All status and interrupt source bits associated with transfers are updated.

- **NAK** (NAK)

  [0] Slave receive transfers generate an ACK at the conclusion of a data transfer.

  [1] Slave receive transfers generate a data NAK (not acknowledge) at the conclusion of a data transfer. The slave is still considered to be addressed.

- **Slave transmit data valid** (STDVAL)

  [0] Data in the transmit FIFO is for master mode transmits and is not allowed to be used during a slave transmit, and the transmit FIFO is treated as if it is empty.

  [1] Data in the transmit FIFO is available for a slave transmission.

- **Slave enable** (SEN)

  [0] The slave is not enabled. No attempt is made to identify a valid address. If cleared during a valid transfer, clock stretching ceases, the serial data line is released, and the current byte is not acknowledged.

  [1] The slave is enabled. Enabling slave and master modes of operation concurrently is allowed.

TWI Slave Mode Address Register (TWI_SLAVE_ADDR)

The TWI_SLAVE_ADDR register holds the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

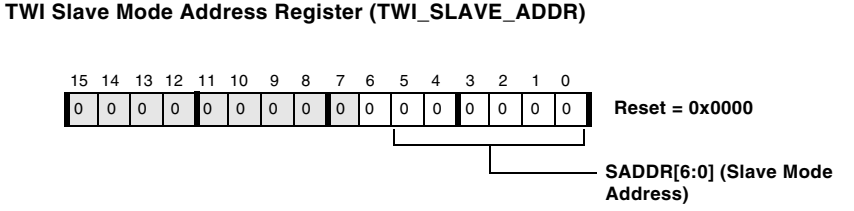

TWI Slave Mode Address Register (TWI_SLAVE_ADDR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SADDR[6:0] (Slave Mode Address)

Figure 12-17. TWI Slave Mode Address Register

TWI Slave Mode Status Register (TWI_SLAVE_STAT)

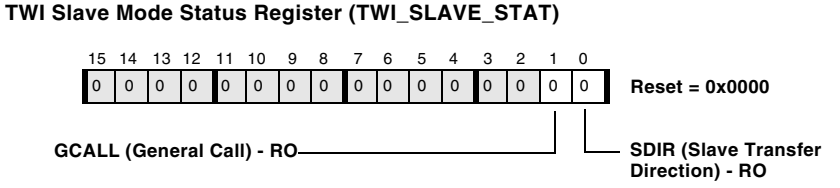

TWI Slave Mode Status Register (TWI_SLAVE_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
GCALL (General Call) - RO
SDIR (Slave Transfer Direction) - RO

Figure 12-18. TWI Slave Mode Status Register

During and at the conclusion of register slave mode transfers, the TWI_SLAVE_STAT register holds information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

- **General call** (GCALL)

This bit self clears if slave mode is disabled (SEN = 0).

[0] At the time of addressing, the address was not determined to be a general call.

[1] At the time of addressing, the address was determined to be a general call.

- **Slave transfer direction** (SDIR)

  This bit self clears if slave mode is disabled (SEN = 0).

  [0] At the time of addressing, the transfer direction was determined to be slave receive.

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

TWI Master Mode Control Register (TWI_MASTER_CTL)

The TWI_MASTER_CTL register controls the logic associated with master mode operation. Bits in this register do not affect slave mode operation and should not be modified to control slave mode functionality.

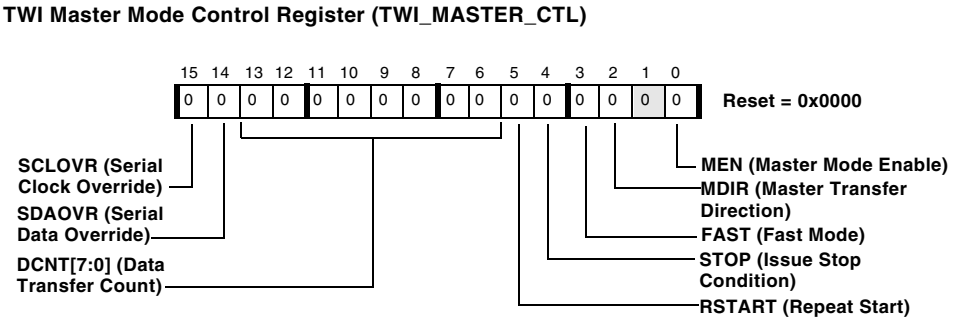

TWI Master Mode Control Register (TWI_MASTER_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SCLOVR (Serial Clock Override)
SDAOVR (Serial Data Override)
DCNT[7:0] (Data Transfer Count)
MEN (Master Mode Enable)
MDIR (Master Transfer Direction)
FAST (Fast Mode)
STOP (Issue Stop Condition)
RSTART (Repeat Start)

Figure 12-19. TWI Master Mode Control Register

Additional information for the TWI_MASTER_CTL register bits includes:

- **Serial clock override** (SCLOVR)

  This bit can be used when direct control of the serial clock line is required. Normal master and slave mode operation should not require override operation.

  [0] Normal serial clock operation under the control of master mode clock generation and slave mode clock stretching logic.

  [1] Serial clock output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- **Serial data (SDA) override** (SDAOVR)

  This bit can be used when direct control of the serial data line is required. Normal master and slave mode operation should not require override operation.

  [0] Normal serial data operation under the control of the transmit shift register and acknowledge logic.

  [1] Serial data output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- **Data transfer count** (DCNT[7:0])

  Indicates the number of data bytes to transfer. As each data word is transferred, DCNT is decremented. When DCNT is 0, a stop condition is generated. Setting DCNT to 0xFF disables the counter. In this transfer mode, data continues to be transferred until it is concluded by setting the STOP bit.

- **Repeat start** (RSTART)

  [0] Transfer concludes with a stop condition.

[1] Issue a repeat start condition at the conclusion of the current transfer (DCNT = 0) and begin the next transfer. The current transfer concludes with updates to the appropriate status and interrupt bits. If errors occurred during the previous transfer, a repeat start does not occur. In the absence of any errors, master enable (MEN) does not self clear on a repeat start.

- **Issue stop condition** (STOP)

  [0] Normal transfer operation.

  [1] The transfer concludes as soon as possible avoiding any error conditions (as if data transfer count had been reached) and at that time the TWI interrupt mask register (TWI_INT_MASK) is updated along with any associated status bits.

- **Fast mode** (FAST)

  [0] Standard mode (up to 100K bits/s) timing specifications in use.

  [1] Fast mode (up to 400K bits/s) timing specifications in use.

- **Master transfer direction** (MDIR)

  [0] The initiated transfer is master transmit.

  [1] The initiated transfer is master receive.

- **Master mode enable** (MEN)

  This bit self clears at the completion of a transfer. This includes transfers terminated due to errors.

  [0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDA, SCL) are no longer driven. Write-1-to-clear status bits are not affected.

[1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

TWI Master Mode Address Register (TWI_MASTER_ADDR)

During the addressing phase of a transfer, the TWI controller, with its master enabled, transmits the contents of the TWI_MASTER_ADDR register. When programming this register, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to this register. For example, if the slave address is b#1010000X, where X is the read/write bit, then TWI_MASTER_ADDR is programmed with b#1010000, which corresponds to 0x50. When sending out the address on the bus, the TWI controller appends the read/write bit as appropriate based on the state of the MDIR bit in the master mode control register.

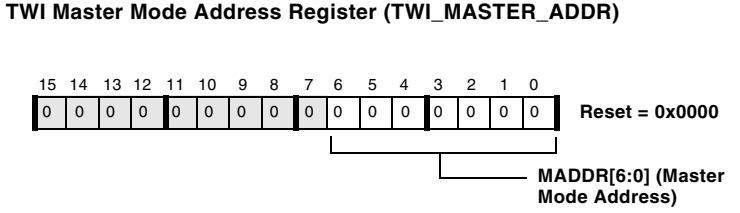

TWI Master Mode Address Register (TWI_MASTER_ADDR)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MADDR[6:0] (Master Mode Address)

Figure 12-20. TWI Master Mode Address Register

TWI Master Mode Status Register (TWI_MASTER_STAT)

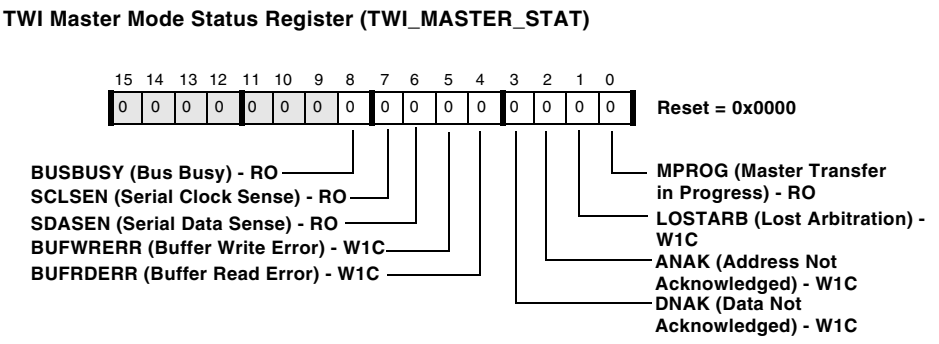

TWI Master Mode Status Register (TWI_MASTER_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
BUSBUSY (Bus Busy) - RO
SCLSEN (Serial Clock Sense) - RO
SDASEN (Serial Data Sense) - RO
BUFWRERR (Buffer Write Error) - W1C
BUFRDERR (Buffer Read Error) - W1C
MPROG (Master Transfer in Progress) - RO
LOSTARB (Lost Arbitration) - W1C
ANAK (Address Not Acknowledged) - W1C
DNAK (Data Not Acknowledged) - W1C

Figure 12-21. TWI Master Mode Status Register

The TWI_MASTER_STAT register holds information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.

Note that—while the SCLSEN bit is set (this could be due to having no pull-up resistor on SCL or another agent is driving SCL low)—the acknowledge bits (ANAK and DNAK) do not update. This result occurs because the acknowledge conditions are sampled during the high phase of SCL.

- **Bus busy** (BUSBUSY)

  Indicates whether the bus is currently busy or free. This indication is not limited to only this device but is for all devices. Upon a start condition, the setting of the register value is delayed due to the input filtering. Upon a stop condition the clearing of the register value occurs after t_{BUF}.

[0] The bus is free. The clock and data bus signals have been inactive for the appropriate bus free time.

[1] The bus is busy. Clock or data activity has been detected.

- **Serial clock sense** (SCLSEN)

  This status bit can be used when direct sensing of the serial clock line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [0] An inactive “one” is being sensed on the serial clock.

  [1] An active “zero” is being sensed on the serial clock. The source of the active driver is not known and can be internal or external.

- **Serial data sense** (SDASEN)

  This status bit can be used when direct sensing of the serial data line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [0] An inactive “one” is currently being sensed on the serial data line.

  [1] An active “zero” is currently being sensed on the serial data line. The source of the active driver is not known and can be internal or external.

- **Buffer write error** (BUFWRERR)

  [0] The current master receive has not detected a receive buffer write error.

  [1] The current master transfer was aborted due to a receive buffer write error. The receive buffer and receive shift register were both full at the same time. This bit is W1C.

- **Buffer read error** (BUFRDERR)

  [0] The current master transmit has not detected a buffer read error.

  [1] The current master transfer was aborted due to a transmit buffer read error. At the time data was required by the transmit shift register the buffer was empty. This bit is W1C.

- **Data not acknowledged** (DNAK)

  [0] The current master receive has not detected a NAK during data transmission.

  [1] The current master transfer was aborted due to the detection of a NAK during data transmission. This bit is W1C.

- **Address not acknowledged** (ANAK)

  [0] The current master transmit has not detected NAK during addressing.

  [1] The current master transfer was aborted due to the detection of a NAK during the address phase of the transfer. This bit is W1C.

- **Lost arbitration** (LOSTARB)

  [0] The current transfer has not lost arbitration with another master.

  [1] The current transfer was aborted due to the loss of arbitration with another master. This bit is W1C.

- **Master transfer in progress** (MPROG)

  [0] Currently no transfer is taking place. This can occur once a transfer is complete or while an enabled master is waiting for an idle bus.

[1] A master transfer is in progress.

TWI FIFO Control Register (TWI_FIFO_CTL)

The TWI_FIFO_CTL register control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

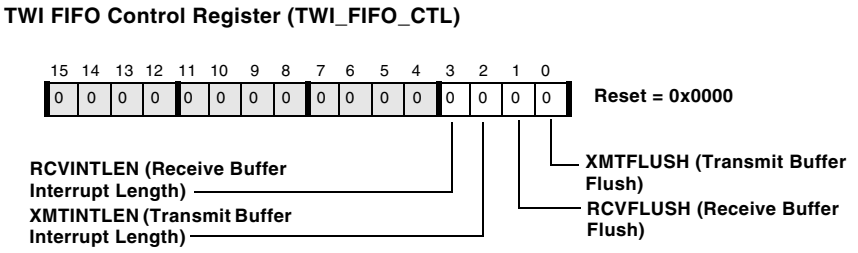

TWI FIFO Control Register (TWI_FIFO_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVINTLEN (Receive Buffer Interrupt Length)
XMTINTLEN (Transmit Buffer Interrupt Length)
XMTFLUSH (Transmit Buffer Flush)
RCVFLUSH (Receive Buffer Flush)

Figure 12-22. TWI FIFO Control Register

Additional information for the TWI_FIFO_CTL register bits includes:

- **Receive buffer interrupt length** (RCVINTLEN)

  This bit determines the rate at which receive buffer interrupts are to be generated. Interrupts may be generated with each byte received or after two bytes are received.

  [0] An interrupt (RCVSERV) is set when RCVSTAT indicates one or two bytes in the FIFO are full (01 or 11).

  [1] An interrupt (RCVSERV) is set when the RCVSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are full (11).

- **Transmit buffer interrupt length** (XMTINTLEN)

  This bit determines the rate at which transmit buffer interrupts are to be generated. Interrupts may be generated with each byte transmitted or after two bytes are transmitted.

[0] An interrupt (XMTSERV) is set when XMTSTAT indicates one or two bytes in the FIFO are empty (01 or 00).

[1] An interrupt (XMTSERV) is set when the XMTSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are empty (00).

- **Receive buffer flush** (RCVFLUSH)

  [0] Normal operation of the receive buffer and its status bits.

  [1] Flush the contents of the receive buffer and update the RCVSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active receive the receive buffer in this state responds to the receive logic as if it is full.

- **Transmit buffer flush** (XMTFLUSH)

  [0] Normal operation of the transmit buffer and its status bits.

  [1] Flush the contents of the transmit buffer and update the XMTSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active transmit the transmit buffer in this state responds as if the transmit buffer is empty.

TWI FIFO Status Register (TWI_FIFO_STAT)

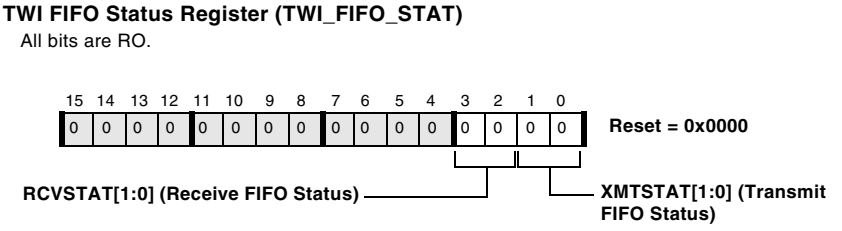

TWI FIFO Status Register (TWI_FIFO_STAT)
All bits are RO.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVSTAT[1:0] (Receive FIFO Status)
XMTSTAT[1:0] (Transmit FIFO Status)

Figure 12-23. TWI FIFO Status Register

TWI FIFO Status

The fields in the TWI_FIFO_STAT register indicate the state of the FIFO buffers' receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

- **Receive FIFO status** (RCVSTAT[1:0])

  The RCVSTAT field is read only. It indicates the number of valid data bytes in the receive FIFO buffer. The status is updated with each FIFO buffer read using the peripheral data bus or write access by the receive shift register. Simultaneous accesses are allowed.

  [00] The FIFO is empty.

  [01] The FIFO contains one byte of data. A single byte peripheral read of the FIFO is allowed.

  [10] Reserved

[11] The FIFO is full and contains two bytes of data. Either a single or double byte peripheral read of the FIFO is allowed.

- **Transmit FIFO status** (`XMTSTAT[1:0]`)

  The `XMTSTAT` field is read only. It indicates the number of valid data bytes in the FIFO buffer. The status is updated with each FIFO buffer write using the peripheral data bus or read access by the transmit shift register. Simultaneous accesses are allowed.

  [00] The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.

  [01] The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  [10] Reserved

  [11] The FIFO is full and contains two bytes of data.

TWI Interrupt Mask Register (TWI_INT_MASK)

The `TWI_INT_MASK` register enables interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the `TWI_INT_STAT` register. Reading and writing the `TWI_INT_MASK` register does not affect the contents of the `TWI_INT_STAT` register.

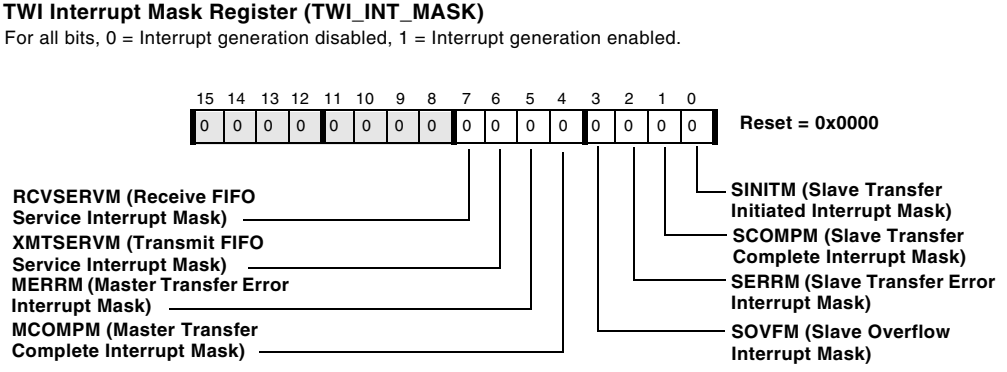

TWI Interrupt Mask Register (TWI_INT_MASK)
For all bits, 0 = Interrupt generation disabled, 1 = Interrupt generation enabled.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVSERVM (Receive FIFO Service Interrupt Mask)
XMTSERVM (Transmit FIFO Service Interrupt Mask)
MERRM (Master Transfer Error Interrupt Mask)
MCOMPM (Master Transfer Complete Interrupt Mask)
SINITM (Slave Transfer Initiated Interrupt Mask)
SCOMPM (Slave Transfer Complete Interrupt Mask)
SERRM (Slave Transfer Error Interrupt Mask)
SOVFM (Slave Overflow Interrupt Mask)

Figure 12-24. TWI Interrupt Mask Register

TWI Interrupt Status Register (TWI_INT_STAT)

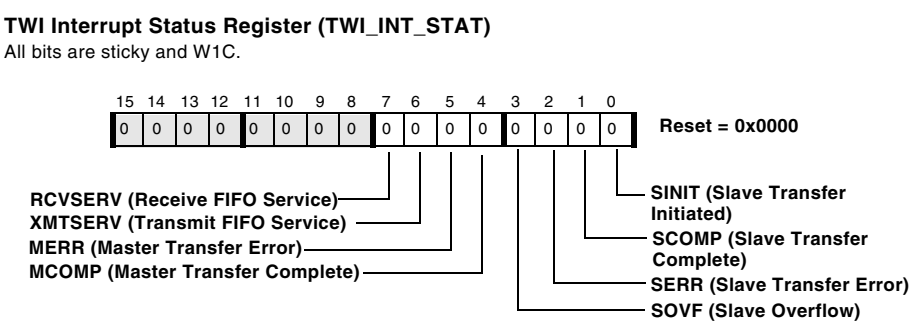

TWI Interrupt Status Register (TWI_INT_STAT)
All bits are sticky and W1C.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVSERV (Receive FIFO Service)
XMTSERV (Transmit FIFO Service)
MERR (Master Transfer Error)
MCOMP (Master Transfer Complete)
SINIT (Slave Transfer Initiated)
SCOMP (Slave Transfer Complete)
SERR (Slave Transfer Error)
SOVF (Slave Overflow)

Figure 12-25. TWI Interrupt Status Register

The `TWI_INT_STAT` register contains information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit by writing a 1 to it.

- **Receive FIFO service** (RCVSERV)

  If RCVINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the RCVSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to or 11.

  [0] No errors have been detected.

  [1] The FIFO does not require servicing or the RCVSTAT field has not changed since this bit was last cleared.

- **Transmit FIFO service** (XMTSERV)

  If XMTINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the XMTSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to 00.

  [0] FIFO does not require servicing or XMTSTAT field has not changed since this bit was last cleared.

  [1] The transmit FIFO buffer has one or two 8-bit locations available to be written.

- **Master transfer error** (MERR)

  [0] No errors have been detected.

  [1] A master error has occurred. The conditions surrounding the error are indicated by the master status register (TWI_MASTER_STAT).

- **Master transfer complete** (MCOMP)

  [0] The completion of a transfer has not been detected.

  [1] The initiated master transfer has completed. In the absence of a repeat start, the bus has been released.

- **Slave overflow** (SOVF)

  [0] No overflow has been detected.

  [1] The slave transfer complete (SCOMP) bit was set at the time a subsequent transfer has acknowledged an address phase. The transfer continues, however, it may be difficult to delineate data of one transfer from another.

- **Slave transfer error** (SERR)

  [0] No errors have been detected.

  [1] A slave error has occurred. A restart or stop condition has occurred during the data receive phase of a transfer.

- **Slave transfer complete** (SCOMP)

  [0] The completion of a transfer has not been detected.

  [1] The transfer is complete and either a stop, or a restart was detected.

- **Slave transfer initiated** (SINIT)

  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.

  [1] The slave has detected an address match and a transfer has been initiated.

TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)

The TWI_XMT_DATA8 register holds an 8-bit data value written into the FIFO buffer. Transmit data is entered into the corresponding transmit buffer in a first-in first-out order. For 16-bit PAB writes, a write access to TWI_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is full, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

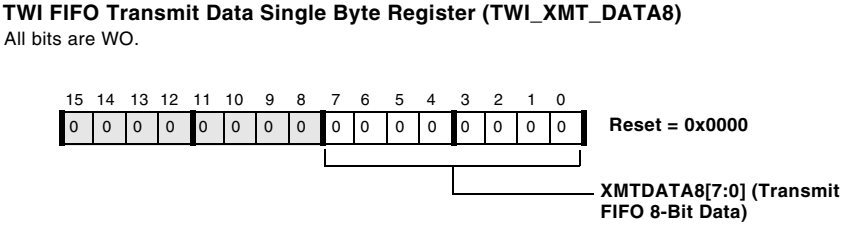

TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)
All bits are WO.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
XMTDATA8[7:0] (Transmit FIFO 8-Bit Data)

Figure 12-26. TWI FIFO Transmit Data Single Byte Register

TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)

The TWI_XMT_DATA16 register holds a 16-bit data value written into the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be d. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access.

The data is written in little endian byte order as shown in Figure 12-27 where byte 0 is the first byte to be transferred and byte 1 is the second byte to be transferred. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the

FIFO buffer is not empty, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

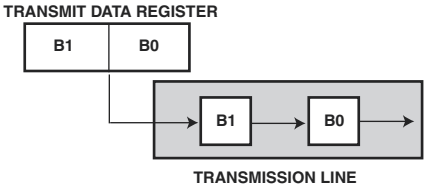


Figure 12-27. Transmit Little Endian Byte Order

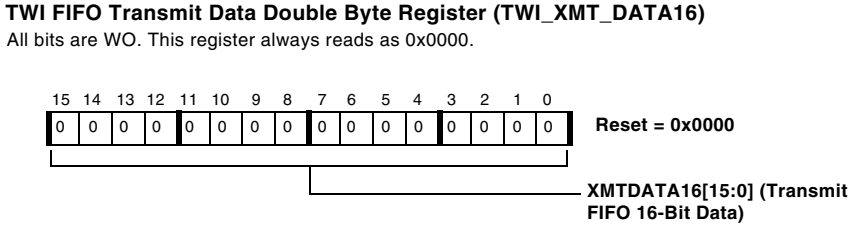

TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)
All bits are WO. This register always reads as 0x0000.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
XMTDATA16[15:0] (Transmit FIFO 16-Bit Data)

Figure 12-28. TWI FIFO Transmit Data Double Byte Register

TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)

The TWI_RCV_DATA8 register holds an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to TWI_RCV_DATA8 will access only one transmit data byte from the FIFO buffer. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is empty, the data is unknown and the FIFO buffer status remains indicating it is empty.

TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)
All bits are RO.

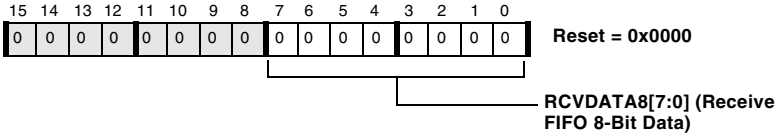


Figure 12-29. TWI FIFO Receive Data Single Byte Register

TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)

The TWI_RCV_DATA16 register holds a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access.

The data is read in little endian byte order as shown in Figure 12-30 where byte 0 is the first byte received and byte 1 is the second byte received. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated to indicate it is empty. If an access is performed while the FIFO buffer is not full, the read data is unknown and the existing FIFO buffer data and its status remains unchanged.

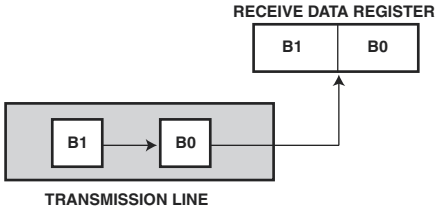


Figure 12-30. Receive Little Endian Byte Order

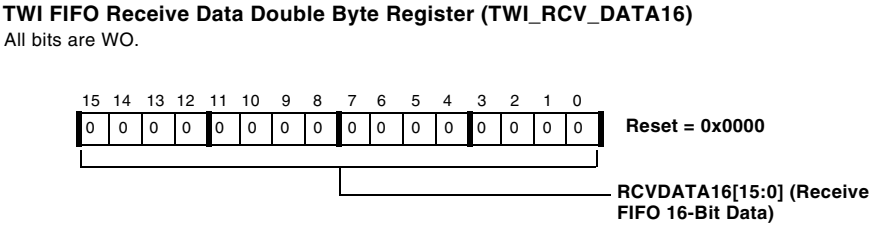

TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)
All bits are WO.
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCVDATA16[15:0] (Receive FIFO 16-Bit Data)

Figure 12-31. TWI FIFO Receive Data Double Byte Register

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.

Master Mode Setup

Listing 12-1 shows how to initiate polled receive and transmit transfers in master mode.

Listing 12-1. Master Mode Receive/Transmit Transfer

```
/*********************************************************
 Macro for the count field of the TWI_MASTER_CTL register
 x can be any value between 0 and 0xFE (254). A value of  0xFF
disables the counter.
*********************************************************/
#define TWICount(x) (DCNT & ((x) << 6))
.section L1_data_b;
.byte TX_file[file_size] = "DATA.hex";
.BYTE RX_CHECK[file_size];
.byte rcvFirstWord[2];
.SECTION program;
```

```
_main:
/*********************************************************
TWI Master Initialization subroutine
*********************************************************/
TWI_INIT:
/*********************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz
P1 points to the base of the system MMRs
*********************************************************/

R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;
/*********************************************************
Set CLKDIV:
For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns)
and an internal time reference of 10 MHz (period = 100 ns):
CLKDIV = 2500 ns / 100 ns = 25
For an SCL with a 30% duty cycle, then CLKLOW = 17 (0x11) and
CLKHI = 8.
*********************************************************/
R5 = CLKHI(0x8) | CLKLOW(0x11) (z);
W[P1 + LO(TWI_CLKDIV)] = R5;
/*********************************************************
enable these signals to generate a TWI interrupt: optional
*********************************************************/
R1 = RCVSERV | XMTSERV | MERR | MCOMP (z);
W[P1 + LO(TWI_INT_MASK)] = R1;
/*********************************************************
The address needs to be shifted one place to the right
e.g., 1010 001x becomes 0101 0001 (0x51) the TWI controller
will actually send out 1010 001x where x is either a 0 for writes
or 1 for reads
```

```
************************************************************/
R6 = 0xBF;
R6 = R6 >> 1;
TWI_INIT.END: W[P1 + LO(TWI_MASTER_ADDR)] = R6;

/******************** END OF TWI INIT **********************/
/************************************************************
Starting the Read transfer
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or SLOW
4. direction of transfer:       MDIR = 1 for reads, MDIR = 0 for
writes
5. Master Enable MEN. This will kick off the master transfer
************************************************************/
R1 = TWICount(0x2) | FAST | MDIR | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
ssync;
/************************************************************
Poll the FIFO Status register to know when
2 bytes have been shifted into the RX FIFO
************************************************************/
Rx_stat:
R1 = W[P1 + LO(TWI_FIFO_STAT)](Z);
R0 = 0xC;
R1 = R1 & R0;
CC = R1 == R0;
IF ! cc jump Rx_stat;
R0 = W[P1 + LO(TWI_RCV_DATA16)](Z); /* Read data from the RX fifo
*/
ssync;
/************************************************************
check that master transfer has completed
```

```
MCOMP will be set when Count reaches zero
************************************************************/
M_COMP:
  R1 = W[P1 + LO(TWI_INT_STAT)](z);
  CC = BITTST (R1, bitpos(MCOMP));
  if ! CC jump M_COMP;
M_COMP.END:  W[P1 + LO(TWI_INT_STAT)] = R1;
/* load the pointer with the address of the transmit buffer */
P2.H = TX_file;
P2.L = TX_file;
/************************************************************
Pre-load the tx FIFO with the first two bytes: this is necessary
to avoid the generation of the Buffer Read Error (BUFRDERR) which
occurs whenever a transmit transfer is initiated while the trans-
mit buffer is empty
************************************************************/
R3 = W[P2++](Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;
/************************************************************
Initiating the Write operation
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or Standard
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. Setting this bit will kick off the transfer
************************************************************/
R1 = TWICount(0xFE) | FAST | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
SSYNC;
/************************************************************
  loop to write data to a TWI slave device P3 times
************************************************************/
```

```
P3 = length(TX_file);
LSETUP (Loop_Start1, Loop_End1) LC0 = P3;
  Loop_Start1:
    /*******************************************************
    check that there's at least one byte location empty in
    the tx fifo
    *******************************************************/
    XMTSERV_Status:
    R1 = W[P1 + LO(TWI_INT_STAT)](z);
    CC = BITTST (R1, bitpos(XMTSERV)); /* test XMTSERV bit */
    if ! CC jump XMTSERV_Status;
    W[P1 + LO(TWI_INT_STAT)] = R1; /* clear status */
    SSYNC;
    /*******************************************************
    write byte into the transmit FIFO
    *******************************************************/
    R3 = B[P2++](Z);
    W[P1 + LO(TWI_XMT_DATA8)] = R3;
  Loop_End1:  SSYNC;
/* check that master transfer has completed */
M_COMP1:
R1 = W[P1 + LO(TWI_INT_STAT)](z);
CC = BITTST (R1, bitpos(MCOMP1));
if ! CC jump M_COMP;
M_COMP1.END:W[P1 + LO(TWI_INT_STAT)] = R1;
idle;
_main.end:
```

Slave Mode Setup

Listing 12-2 shows how to configure the slave for interrupt based transfers. The interrupts are serviced in the subroutine `_TWI_ISR` shown in Listing 12-3.

Listing 12-2. Slave Mode Setup

```
#include <defBF527.h>
/*BF527 is used here as an example—change as appropriate.*/
#include "startup.h"
#define file_size 254
#define SYSMMR_BASE 0xFFC00000
#define COREMMR_BASE 0xFFE00000
.GLOBAL _main;
.EXTERN _TWI_ISR;
.section L1_data_b;
.BYTE TWI_RX[file_size];
.BYTE TWI_TX[file_size] = "transmit.dat";
.section L1_code;
_main:
/*************************************************************
TWI Slave Initialization subroutine
*************************************************************/
TWI_SLAVE_INIT:
/*************************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz
P1 points to the base of the system MMRs
P0 points to the base of the core MMRs
*************************************************************/
R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;
/*************************************************************
Slave address
program the address to which this slave will respond to.
this is an arbitrary 7-bit value
*************************************************************/
R1 = 0x5F;
```

```
W[P1 + LO(TWI_SLAVE_ADDR)] = R1;
/*********************************************************
Pre-load the TX FIFO with the first two bytes to be transmitted
in the event the slave is addressed and a transmit is required
*********************************************************/
R3=0xB537(Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;
/*********************************************************
FIFO Control determines whether an interrupt is generated
for every byte transferred or for every two bytes.
A value of zero which is the default, allows for single byte
events to generate interrupts
*********************************************************/
R1 = 0;
 W[P1 + LO(TWI_FIFO_CTL)] = R1;
/*********************************************************
enable these signals to generate a TWI interrupt
*********************************************************/
R1 = RCVSERV | XMTSERV | SOVF | SERR | SCOMP | SINIT (z);
W[P1 + LO(TWI_INT_MASK)] = R1;
/*********************************************************
Enable the TWI Slave
Program the Slave Control register with:
1. Slave transmit data valid (STDVAL) set so that the contents of
the TX FIFO can be used by this slave when a master requests data
from it.
2. Slave Enable SEN to enable Slave functionality
*********************************************************/
R1 = STDVAL | SEN;
W[P1 + LO(TWI_SLAVE_CTL)] = R1;
TWI_SLAVE_INIT.END:
P2.H = HI(TWI_RX);
P2.L = LO(TWI_RX);
```

```
P4.H = HI(TWI_TX);
P4.L = LO(TWI_TX);
/*********************************************************
Remap the vector table pointer from the default __I10HANDLER
to the new _TWI_ISR interrupt service routine
*********************************************************/
R1.H = HI(_TWI_ISR);
R1.L = LO(_TWI_ISR);
[P0 + LO(EVT10)] = R1;   /* note that P0 points to the base of
the core MMR registers */
/*********************************************************
ENABLE TWI generate to interrupts at the system level
*********************************************************/
R1 = [P1 + LO(SIC_IMASK)];
BITSET(R1,BITPOS(IRQ_TWI));
[P1 + LO(SIC_IMASK)] = R1;
/*********************************************************
ENABLE TWI to generate interrupts at the core level
*********************************************************/
R1 = [P0 + LO(IMASK)];
BITSET(R1,BITPOS(EVT_IVG10));
[P0 + LO(IMASK)] = R1;
/*********************************************************
 wait for interrupts
*********************************************************/
idle;

_main.END:
```

Listing 12-3. TWI Slave Interrupt Service Routine

```
/*********************************************************
```

```
Function:  _ TWI_ISRDescription:  This ISR is executed when the
TWI controller detects a slave initiated transfer. After an
interrupt is serviced, its corresponding bit is cleared in the
TWI_INT_STAT register. This done by writing a 1 to the particular
bit position. All bits are write 1 to clear.
*************************************************************/
#include <defBF527.h>
/*BF527 is used here as an example—change as appropriate.*/
.GLOBAL _TWI_ISR;

.section L1_code;
_TWI_ISR:
/************************************************************
read the source of the interrupt
*************************************************************/
R1 = W[P1 + LO(TWI_INT_STAT)](z);
/************************************************************
Slave Transfer Initiated
*************************************************************/
CC = BITTST(R1, BITPOS(SINIT));
if ! CC JUMP RECEIVE;
R0 = SINIT (Z);
W[P1 + LO(TWI_INT_STAT)] = R0;  /* clear interrupt source bit */
ssync;
/************************************************************
Receive service
*************************************************************/
RECEIVE:
CC = BITTST(R1, BITPOS(RCVSERV));
if ! CC JUMP TRANSMIT;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z);   /* read data */
B[P2++] = R0;   /* store bytes into a buffer pointed to by P2 */
R0 = RCVSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /*clear interrupt source bit */
```

```
ssync;
JUMP _TWI_ISR.END; /* exit */
/*********************************************************
Transmit service
*********************************************************/
TRANSMIT:
CC = BITTST(R1, BITPOS(XMTSERV));
if ! CC JUMP SlaveError;
R0 = B[P4++](Z);
W[P1 + LO(TWI_XMT_DATA8)] = R0;
R0 = XMTSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */
/*********************************************************
slave transfer error
*********************************************************/
SlaveError:
CC = BITTST(R1, BITPOS(SERR));
if ! CC JUMP SlaveOverflow;
R0 = SERR(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END;   /* exit */
/*********************************************************
slave overflow
*********************************************************/
SlaveOverflow:
CC = BITTST(R1, BITPOS(SOVF));
if !CC JUMP SlaveTransferComplete;
R0 = SOVF(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END;   /* exit */
```

```
/*********************************************************
 slave transfer complete
*********************************************************/
SlaveTransferComplete:
CC = BITTST(R1, BITPOS(SCOMP));
if !CC JUMP _TWI_ISR.END;
R0 = SCOMP(Z);
W[P1 + LO(TWI_INT_STAT)] = R0;   /* clear interrupt source bit */
ssync;
/* Transfer complete read receive FIFO buffer and set/clear sema-
phores etc.... */
R0 = W[P1 + LO(TWI_FIFO_STAT)](z);
CC = BITTST(R0,BITPOS(RCV_HALF));    /* BIT 2 indicates whether
there's a byte in the FIFO or not */
if !CC JUMP _TWI_ISR.END;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z);  /* read data */
B[P2++] = R0;   /* store bytes into a buffer pointed to by P2 */
_TWI_ISR.END:RTI;
```

Electrical Specifications

All logic complies with the Electrical Specification outlined in the *Philips I^2C Bus Specification version 2.1* dated January 2000.

Unique Information for the ADSP-BF59x Processor

None.

13 SPI-COMPATIBLE PORT CONTROLLER

This chapter describes the serial peripheral interface (SPI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of SPIs for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For SPI DMA channel assignments, refer to Table 5-9 on page 5-103 in Chapter 5, "Direct Memory Access".

For SPI interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

To determine how each of the SPIs is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each SPI, refer to Chapter A, "System MMR Assignments".

SPI behavior for the ADSP-BF59x that differs from the general information in this chapter can be found in the section "Unique Information for the ADSP-BF59x Processor" on page 13-52.

Overview

The SPI port provides an I/O interface to a wide variety of SPI-compatible peripheral devices.

With a range of configurable options, the SPI port provides a glueless hardware interface with other SPI-compatible devices. SPI is a four-wire interface consisting of two data signals, a device select signal, and a clock signal. SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI-compatible peripheral implementation also supports programmable bit rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Features

The SPI includes these features:

- Full duplex, synchronous serial interface
- Supports 8- or 16-bit word sizes
- Programmable baud rate, clock phase, and polarity
- Supports multimaster environments
- Integrated DMA controller
- Double-buffered transmitter and receiver
- One SPI device select input and multiple chip select outputs
- Programmable shift direction of MSB or LSB first
- Interrupt generation on mode fault, overflow, and underflow
- Shadow register to aid debugging

Typical SPI-compatible peripheral devices that can be used to interface to the SPI-compatible interface include:

- Other CPUs or microcontrollers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays
- Shift registers
- FPGAs with SPI emulation

Interface Overview

Figure 13-1 on page 13-4 provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCK rate, to and from other SPI devices. SPI data is transmitted and received at the same time through the use of a shift register. When an SPI transfer occurs, data is simultaneously transmitted (shifted serially out of the shift register) as new data is received (shifted serially into the other end of the same shift register). The SCK synchronizes the shifting and sampling of the data on the two serial data pins.

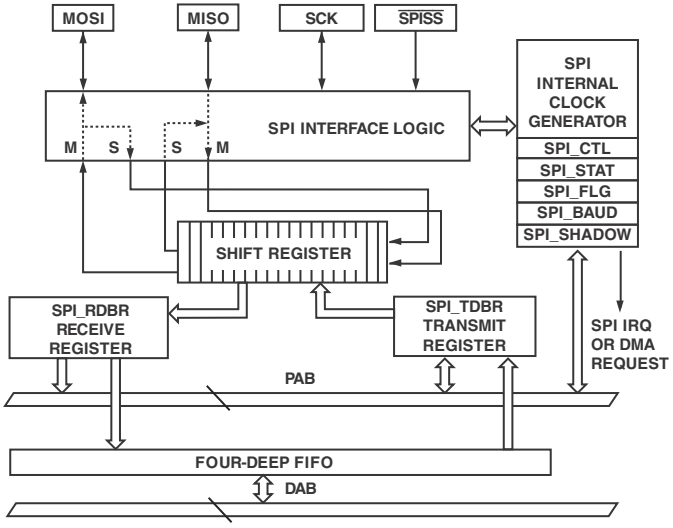


Figure 13-1. SPI Block Diagram

External Interface

SPI Clock Signal (SCK)

The SCK signal is the serial clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of bit rates. The SCK signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The SCK is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the SPISS input is driven inactive (high).

The SCK is used to shift out and shift in the data driven on the MISO and MOSI lines. Clock polarity and clock phase relative to data are programmable in the SPI_CTL register and define the transfer format.

Master-Out, Slave-In (MOSI) Signal

The master-out, slave-in (MOSI) signal is one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSI pin transmits data out. If the processor is configured as a slave, the MOSI pin receives data in. In an SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s).

Master-In, Slave-Out (MISO) Signal

The master-in, slave-out (MISO) signal is one of the bidirectional I/O data pins. If the processor is configured as a master, the MISO pin receives data in. If the processor is configured as a slave, the MISO pin transmits data out. In an SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master.

Only one slave is allowed to transmit data at any given time.

The SPI configuration example in Figure 13-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

The processor can be booted through its SPI interface to allow user application code and data to be downloaded before runtime.

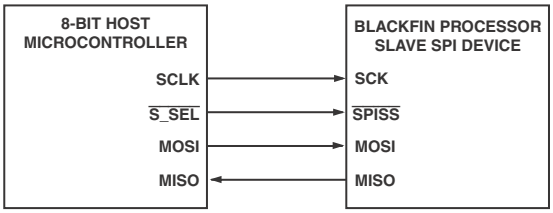


Figure 13-2. Blackfin Processor as Slave SPI Device

SPI Slave Select Input Signal (SPISS)

The SPISS signal is the SPI slave select input signal. This is an active-low signal used to enable a processor when it is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, it can act as an error signal input in a multimaster environment. In multimaster mode, if the SPISS input signal of a master is asserted (driven low), and the PSSE bit in the SPI_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the SCK period. See Figure 13-3 on page 13-7. The minimum time between successive word transfers (T4) is two SCK periods. This is measured from the last active edge of SCK of one word to the first active edge of SCK of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

For a master device with CPHA = 0, the slave select output is inactive (high) for at least one-half the SCK period. In this case, T1 and T2 will each always be equal to one-half the SCK period.

SPI Slave Select Enable Output Signals

When operating in master mode, Blackfin processors may use any GPIO pin to enable individual SPI slave devices by software. In addition, the SPI

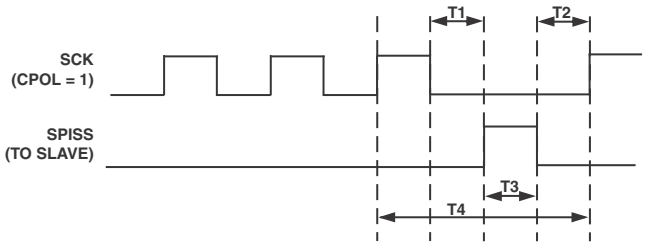


Figure 13-3. SPI Timing

module provides hardware support to generate up to seven slave select enable signals automatically (depending upon the configuration of the specific processor). See Figure 13-14 on page 13-37 for details.

These signals are always active low in the SPI protocol. Since the respective pins are not driven during reset, it is recommended to pull them up by a resistor.

If enabled as a master, the SPI uses the SPI_FLG register to enable general-purpose port pins to be used as individual slave select lines. Before manipulating this register, the port pins that are to be used as SPI slave-select outputs must first be configured as such. To work as SPI output pins, the port pins must be enabled for use by SPI in the appropriate PORT_MUX register.

In slave mode, the SPI_FLG bits have no effect, and each SPI uses the SPISS input as a slave select. Just as in the master mode case, the port pin associated with SPISS must first be configured appropriately before use. Figure 13-14 on page 13-37 shows the SPI_FLG register diagram.

Slave Select Inputs

If the SPI is in slave mode, SPISS acts as the slave select input. When enabled as a master, SPISS can serve as an error detection input for the SPI in a multimaster environment. The PSSE bit in SPI_CTL enables this

feature. When PSSE = 1, the SPISS input is the master mode error input. Otherwise, SPISS is ignored.

Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPI_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including a master processor equipped with seven slave selects, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI is the master. The seven port pins that can be configured as SPI master mode slave-select output pins can be connected to each of the slave SPI device's SPISS pins. In this configuration, the FLSx bits in SPI_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.
2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected through SPI ports can be other processors.
3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in some other microcontrollers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 13-4 shows one processor as a master with three processors (or other SPI-compatible devices) as slaves.

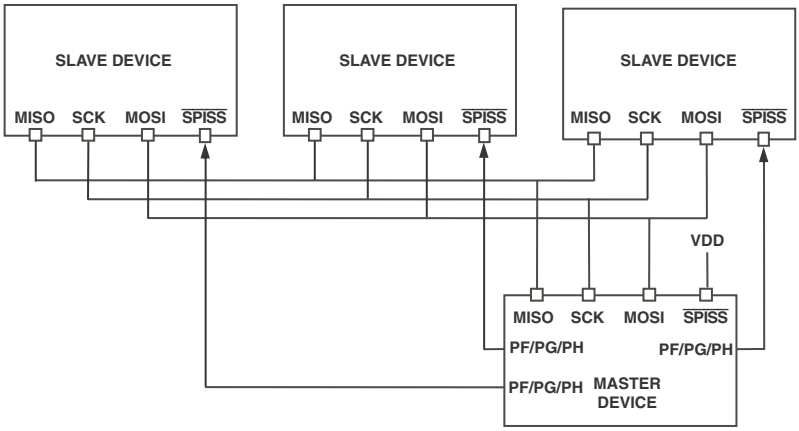


Figure 13-4. Single-Master, Multiple-Slave Configuration

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.

The SPIF bit in the SPI_STAT register is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit in the SPI_STAT register are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA

FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

Internal Interfaces

The SPI has dedicated connections to the processor's peripheral bus (PAB) and DAB.

The low-latency PAB bus is used to map the SPI resources into the system MMR space. For PAB accesses to SPI MMRs, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the peripheral bus are two SCLK cycles.

The DAB bus provides a means for DMA SPI transfers to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory. The SPI peripheral, as a DMA master, is capable of sourcing DMA accesses. The arbitration policy for access to the DAB is described in the *Chip Bus Hierarchy* chapter.

DMA Functionality

The SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data will essentially be ignored.

When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DAB.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA

FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred and the SPI can be disabled or enabled for another mode.

The four-word FIFO is cleared when the SPI port is disabled.

Description of Operation

The following sections describe the operation of the SPI.

SPI Transfer Protocols

The SPI protocol supports four different combinations of serial clock phase and polarity (SPI modes 0, 1, 2, 3). These combinations are selected using the CPOL and CPHA bits in SPI_CTL as shown in Figure 13-5 on page 13-11.

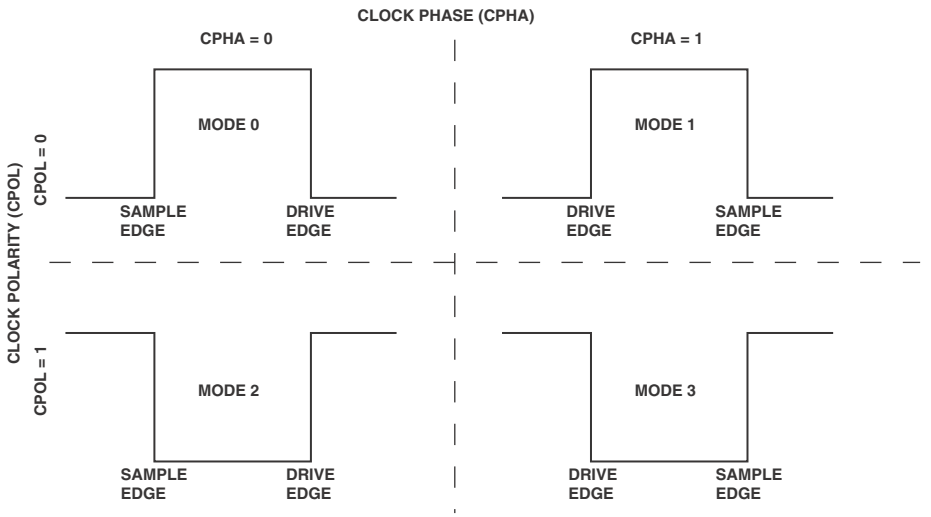


Figure 13-5. SPI Modes of Operation

Figure 13-6 on page 13-13 and Figure 13-7 on page 13-13 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are shown for SCK—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SCK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SCK signal is generated by the master, and the SPISS signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE = 0) with the most significant bit (MSB) first (LSBF = 0). Any combination of the SIZE and LSBF bits of SPI_CTL is allowed. For example, a 16-bit transfer with the least significant bit (LSB) first is another possible configuration.

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When CPHA = 0, the slave select line, SPISS, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, SPISS may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software through manipulation of the SPI_FLG register.

Figure 13-6 shows the SPI transfer protocol for CPHA = 0. Note SCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

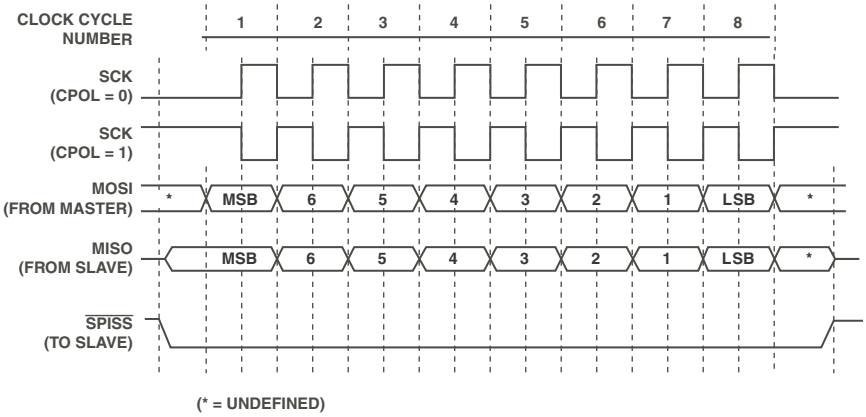


Figure 13-6. SPI Transfer Protocol for CPHA = 0

Figure 13-7 shows the SPI transfer protocol for CPHA = 1. Note SCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

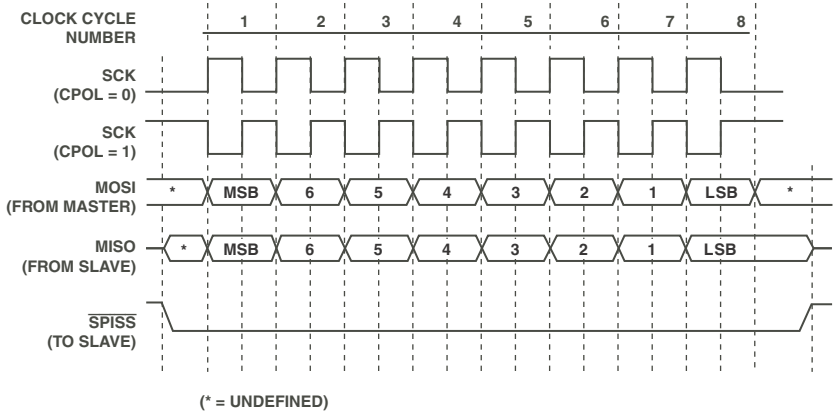


Figure 13-7. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI can be used in single master as well as multimaster environments. The MOSI, MISO, and the SCK signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the broadcast mode has been selected. In broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in transmit mode driving the MISO line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSI and MISO) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSI and MISO pins when this option is selected.

The WOM bit in the SPI_CTL register controls this option. When WOM is set and the SPI is configured as a master, the MOSI pin is three-stated when the data driven out on MOSI is a logic high. The MOSI pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISO pin is three-stated if the data driven out on MISO is a logic high.

During SPI data transfers, one SPI device acts as the SPI link master, where it controls the data flow by generating the SPI serial clock and asserting the SPI device select signal (SPISS). The other SPI device acts as

the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as broadcast mode). However, only one slave may drive its output to write data back to the master at any given time. This must be enforced in broadcast mode, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected through their SPI ports, all MOSI pins are connected together, all MISO pins are connected together, and all SCK pins are connected together.

For a multislave environment, the processor can make use of up to seven programmable flags that are dedicated SPI slave select signals for the SPI slave devices.

At reset, the SPI is disabled and configured as a slave.

Clock Signals

The SCK signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value in the SPI_BAUD register. For slave devices, the value in SPI_BAUD is ignored. When the SPI device is a master, SCK is an output signal. When the SPI is a slave, SCK is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The SCK signal is used to shift out and shift in the data driven onto the MISO and MOSI lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and

clock phase relative to data are programmable in the SPI_CTL register and define the transfer format. See Figure 13-5 on page 13-11.

Interrupt Output

The SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the TIMOD field in the SPI_CTL register. In DMA mode (TIMOD = b#1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = b#11) or read from (TIMOD = b#10). In non-DMA mode (TIMOD = 0X), a data interrupt is generated when the SPI_TDBR register is ready to be written to (TIMOD = b#01) or when the SPI_RDBR register is ready to be read from (TIMOD = b#00).

An SPI error interrupt is generated in a master when a mode fault error occurs, in both DMA and non-DMA modes. An error interrupt can also be generated in DMA mode when there is an underflow (TXE when TIMOD = b#11) or an overflow (RBSY when TIMOD = b#10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPI_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI Control (SPI_CTL) Register" on page 13-35.

Functional Description

The following sections describe the functional operation of the SPI.

Master Mode Operation (Non-DMA)

When the SPI is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to the appropriate port register(s) to properly configure the SPI interface for master mode operation.The required pins are configured for SPI use as slave-select outputs.
2. The core writes to SPI_FLG, setting one or more of the SPI flag select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.
3. The core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.
4. If the CPHA bit in the SPI_CTL register = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI_FLG.
5. The TIMOD bits in SPI_CTL determine the SPI transfer initiate mode. The transfer on the SPI link begins upon either a data write by the core to the SPI_TDBR register or a data read of the SPI_RDBR register.
6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. Before a shift, the shift register is loaded with the contents of the SPI_TDBR register. At the end of the transfer, the contents of the shift register are loaded into the SPI_RDBR register.
7. With each new transfer initiate command, the SPI continues to send and receive words, according to the SPI transfer initiate mode.

See Figure 13-8 on page 13-29 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL.

If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits zeros on the MOSI pin. One word is transmitted for each new transfer initiate command. If SZ = 0 and the transmit buffer is empty, the device repeatedly transmits the last word it transmitted before the transmit buffer became empty.

If GM = 1 and the receive buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Transfer Initiation From Master (Transfer Modes)

When a device is enabled as a master, the initiation of a transfer is defined by the two TIMOD bits of SPI_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of the SPI_RDBR register or a write to the SPI_TDBR register. This is summarized in Table 13-1.

If the SPI port is enabled with TIMOD = b#01 or TIMOD = b#11, the hardware immediately issues a first interrupt or DMA request.

Table 13-1. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
b#00	Transmit and receive	Initiate new single word transfer upon read of SPI_RDBR and previous transfer completed.	Interrupt is active when the receive buffer is full. Read of SPI_RDBR clears interrupt.
b#01	Transmit and receive	Initiate new single word transfer upon write to SPI_TDBR and previous transfer completed.	Interrupt is active when the transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
b#10	Receive with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA read of SPI_RDBR, and last transfer completed.	Request DMA reads as long as the SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA write to SPI_TDBR, and last transfer completed.	Request DMA writes as long as the SPI DMA FIFO is not full.

Slave Mode Operation (Non-DMA)

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the SPISS select signal to the active state (low), or by the first active edge of the clock (SCK), depending on the state of the CPHA bit in the SPI_CTL register.

These steps illustrate SPI operation in the slave mode:

1. The core writes to the appropriate port register(s) to properly configure the SPI for slave mode operation.
2. The core writes to SPI_CTL to define the mode of the serial link to be the same as the mode set up in the SPI master.
3. To prepare for the data transfer, the core writes data to be transmitted into SPI_TDBR.
4. Once the SPISS falling edge is detected, the slave starts shifting data out on MISO and in from MOSI on SCK edges, depending upon the states of CPHA and CPOL.
5. Reception/transmission continues until SPISS is released or until the slave has received the proper number of clock cycles.
6. The slave device continues to receive/transmit with each new falling edge transition on SPISS and/or SCK clock edge.

See Figure 13-8 on page 13-29 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits zeros on the MISO pin. If SZ = 0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the receive buffer is full, the incoming data is discarded, and the SPI_RDBR register is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 13-2 are necessary to prepare the device for a new transfer.

Table 13-2. Transfer Preparation

TIMOD	Function	Action, Interrupt
b#00	Transmit and receive	Interrupt is active when the receive buffer is full. Read of SPI_RDBR clears interrupt.
b#01	Transmit and receive	Interrupt is active when the transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
b#10	Receive with DMA	Request DMA reads as long as SPI DMA FIFO is not empty.
b#11	Transmit with DMA	Request DMA writes as long as SPI DMA FIFO is not full.

Programming Model

The following sections describe the SPI programming model.

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, which CPHA mode is selected, and which transfer initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPI_TDBR is written to or SPI_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCK signal remains inactive for the first half of the first cycle of SCK. For a slave with CPHA = 0, the transfer starts as soon as the SPISS input goes low.

For CPHA = 1, a transfer starts with the first active edge of SCK for both slave and master devices. For a master device, a transfer is considered

finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCK.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPI_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SCK. For a slave SPI, RXS is set shortly after the last SCK edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPI_RDBR is full (TIMOD = b#00), the interrupt goes active one SCLK cycle after RXS is set. When not relying on this interrupt, the end of a transfer can be detected by polling the RXS bit.

To maintain software compatibility with other SPI devices, the SPIF bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, SPIF is cleared shortly after the start of a transfer (SPISS going low for CPHA = 0, first active edge of SCK on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a transfer (either by writing the SPI_TDBR or reading the SPI_RDBR, depending on TIMOD), and is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPI_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPI_RDBR, because of the latency. Therefore, for SPI_BAUD = 2 or SPI_BAUD = 3, RXS must be set before SPIF to read SPI_RDBR. For larger SPI_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the TIMOD = b#00 mode may be the best operation option. In this mode,

software performs a dummy read from the SPI_RDBR register to initiate the first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the SPI_TDBR register before performing the dummy read. If the transmitted value is arbitrary, it is good practice to set the SZ bit in the SPI_CTL register to ensure zero data is transmitted rather than random values. When receiving the last word of an SPI stream, software should ensure that the read from the SPI_RDBR register does not initiate another transfer. It is recommended that the SPI port be disabled before the final SPI_RDBR read access. Reading the SPI_SHADOW register is not sufficient, as it does not clear the interrupt request.

In master mode with the CPHA bit set, software should manually assert the required slave select signal before starting the transaction. After all data has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine only when operating in TIMOD = b#00 or TIMOD = b#10 mode. With TIMOD = b#01 or TIMOD = b#11, the interrupt is requested while the transfer is still in progress.

Master Mode DMA Operation

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI interface operates as follows.

1. The core writes to the appropriate port register(s) to properly configure the SPI for master mode operation. The appropriate pins can be configured for SPI use as slave-select outputs.
2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and to configure the necessary work units, access direction, word count, and so on. For more information, see the *Direct Memory Access* chapter.

3. The processor core writes to the SPI_FLG register, setting one or more of the SPI flag select bits (FLSx).

4. The processor core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either "receive with DMA" (TIMOD = b#10) or "transmit with DMA" (TIMOD = b#11) mode.

5. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPI_TDBR register, it initiates a transfer on the SPI link.

6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. For receive transfers, the value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. For transmit transfers, the value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

7. In receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from "1" to "0". The SPI continues receiving words until SPI DMA mode is disabled.

In transmit mode, as long as there is room in the SPI DMA FIFO (the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from “1” to “0”. The SPI continues transmitting words until the SPI DMA FIFO is empty.

See Figure 13-9 on page 13-30 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit in the SPI_CTL register. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0, and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and TXE is set). If SZ = 1, the device repeatedly transmits zeros on the MOSI pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, the master SPI initiates a word transfer only when there is data in the DMA FIFO. If the DMA FIFO is empty, the SPI waits for the DMA engine to write to the DMA FIFO before starting the transfer. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode. The TXE underrun condition cannot happen in this mode (master DMA TX mode), because the master SPI will not initiate a transfer if there is no data in the DMA FIFO.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10).

A master SPI DMA sequence may involve back-to-back transmission and/or reception of multiple DMA work units. The SPI controller supports such a sequence with minimal core interaction.

Slave Mode DMA Operation

When enabled as a slave with the DMA engine configured to transmit or receive data, the start of a transfer is triggered by a transition of the SPISS signal to the active-low state or by the first active edge of SCK, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The core writes to the appropriate port register(s) to properly configure the SPI for slave mode operation.
2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and configure the necessary work units, access direction, word count, and so on. For more information, see the *Direct Memory Access* chapter.
3. The processor core writes to the SPI_CTL register to define the mode of the serial link to be the same as the mode set up in the SPI master. The TIMOD field will be configured to select either “receive with DMA” (TIMOD = b#10) or “transmit with DMA” (TIMOD = b#11) mode.
4. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the shift register is loaded into the SPI_RDBR register at the end

of the transfer. As the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPI_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on SCK edges. The value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

5. In receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from “1” to “0”. The SPI slave continues receiving words on SCK edges as long as the slave select input is active.

   In transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from “1” to “0”. The SPI slave continues transmitting words on SCK edges as long as the slave select input is active.

See Figure 13-9 on page 13-30 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit in the SPI_CTL register. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MOSI pin, overwriting the

older data in the SPI_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits zeros on the MISO pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits zeros on the MISO pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = b#10), or when the DMA FIFO is not full (when TIMOD = b#11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = b#10), or when there is a TXE underflow error condition (when TIMOD = b#11).

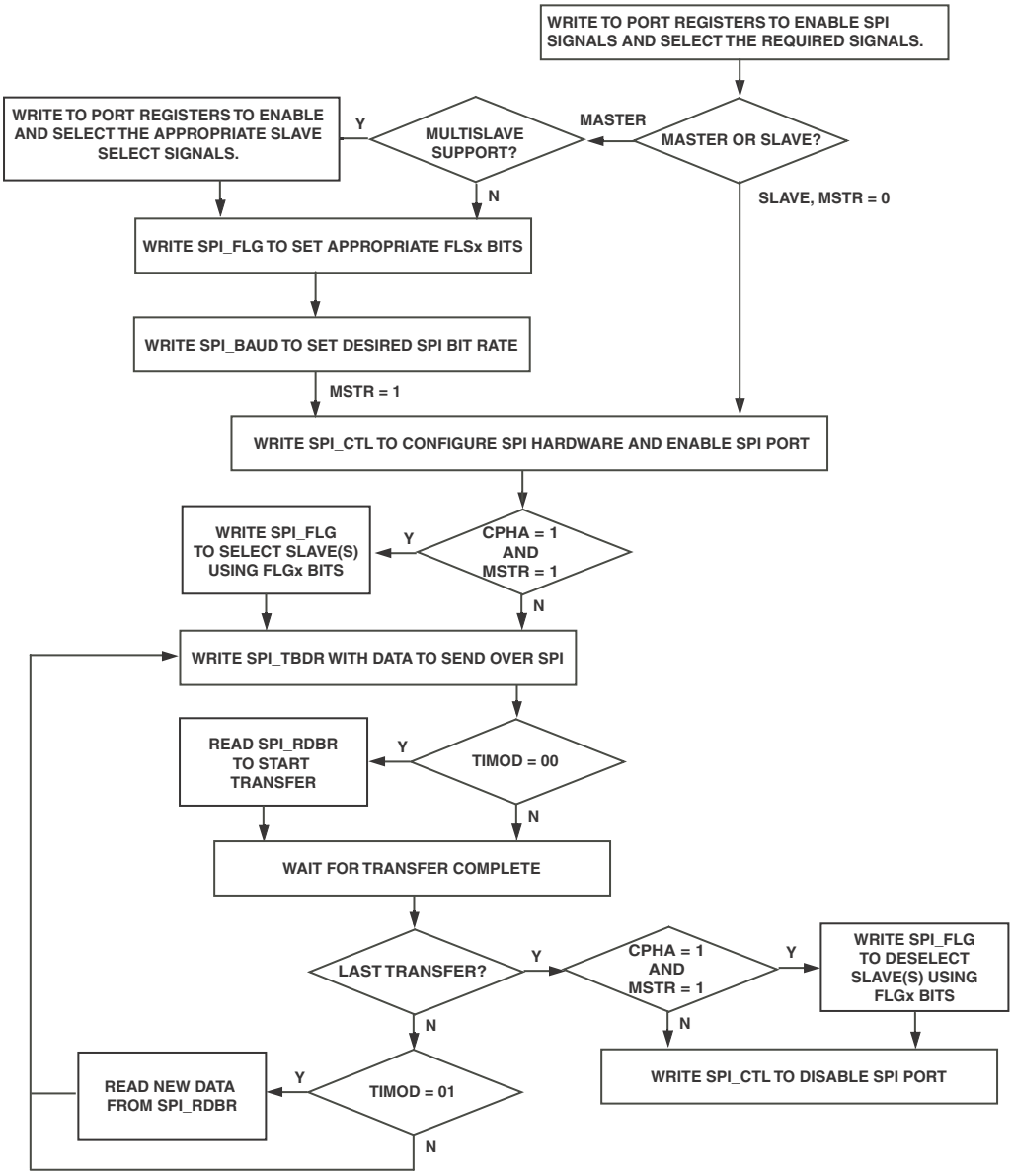

WRITE TO PORT REGISTERS TO ENABLE SPI SIGNALS AND SELECT THE REQUIRED SIGNALS.
MASTER OR SLAVE?
MASTER
SLAVE, MSTR = 0
MULTISLAVE SUPPORT?
Y
N
WRITE TO PORT REGISTERS TO ENABLE AND SELECT THE APPROPRIATE SLAVE SELECT SIGNALS.
WRITE SPI_FLG TO SET APPROPRIATE FLSx BITS
WRITE SPI_BAUD TO SET DESIRED SPI BIT RATE
MSTR = 1
WRITE SPI_CTL TO CONFIGURE SPI HARDWARE AND ENABLE SPI PORT
CPHA = 1 AND MSTR = 1
WRITE SPI_FLG TO SELECT SLAVE(S) USING FLGx BITS
WRITE SPI_TBDR WITH DATA TO SEND OVER SPI
TIMOD = 00
READ SPI_RDBR TO START TRANSFER
WAIT FOR TRANSFER COMPLETE
LAST TRANSFER?
CPHA = 1 AND MSTR = 1
WRITE SPI_FLG TO DESELECT SLAVE(S) USING FLGx BITS
WRITE SPI_CTL TO DISABLE SPI PORT
TIMOD = 01
READ NEW DATA FROM SPI_RDBR

Figure 13-8. Core-Driven SPI Flow Chart

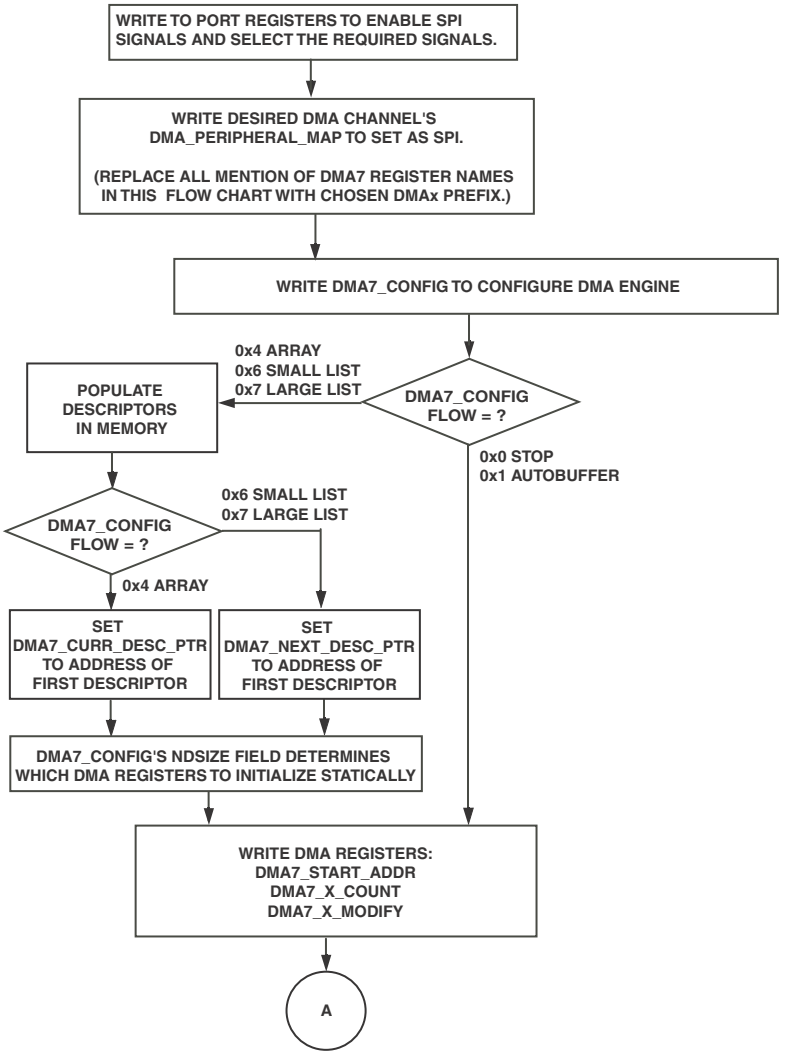

WRITE TO PORT REGISTERS TO ENABLE SPI SIGNALS AND SELECT THE REQUIRED SIGNALS.
WRITE DESIRED DMA CHANNEL'S DMA_PERIPHERAL_MAP TO SET AS SPI.
(REPLACE ALL MENTION OF DMA7 REGISTER NAMES IN THIS FLOW CHART WITH CHOSEN DMAx PREFIX.)
WRITE DMA7_CONFIG TO CONFIGURE DMA ENGINE
0x4 ARRAY
0x6 SMALL LIST
0x7 LARGE LIST
DMA7_CONFIG FLOW = ?
POPULATE DESCRIPTORS IN MEMORY
0x0 STOP
0x1 AUTOBUFFER
0x6 SMALL LIST
0x7 LARGE LIST
DMA7_CONFIG FLOW = ?
0x4 ARRAY
SET DMA7_CURR_DESC_PTR TO ADDRESS OF FIRST DESCRIPTOR
SET DMA7_NEXT_DESC_PTR TO ADDRESS OF FIRST DESCRIPTOR
DMA7_CONFIG'S NDSIZE FIELD DETERMINES WHICH DMA REGISTERS TO INITIALIZE STATICALLY
WRITE DMA REGISTERS:
DMA7_START_ADDR
DMA7_X_COUNT
DMA7_X_MODIFY
A

Figure 13-9. SPI DMA Flow Chart (Part 1 of 3)

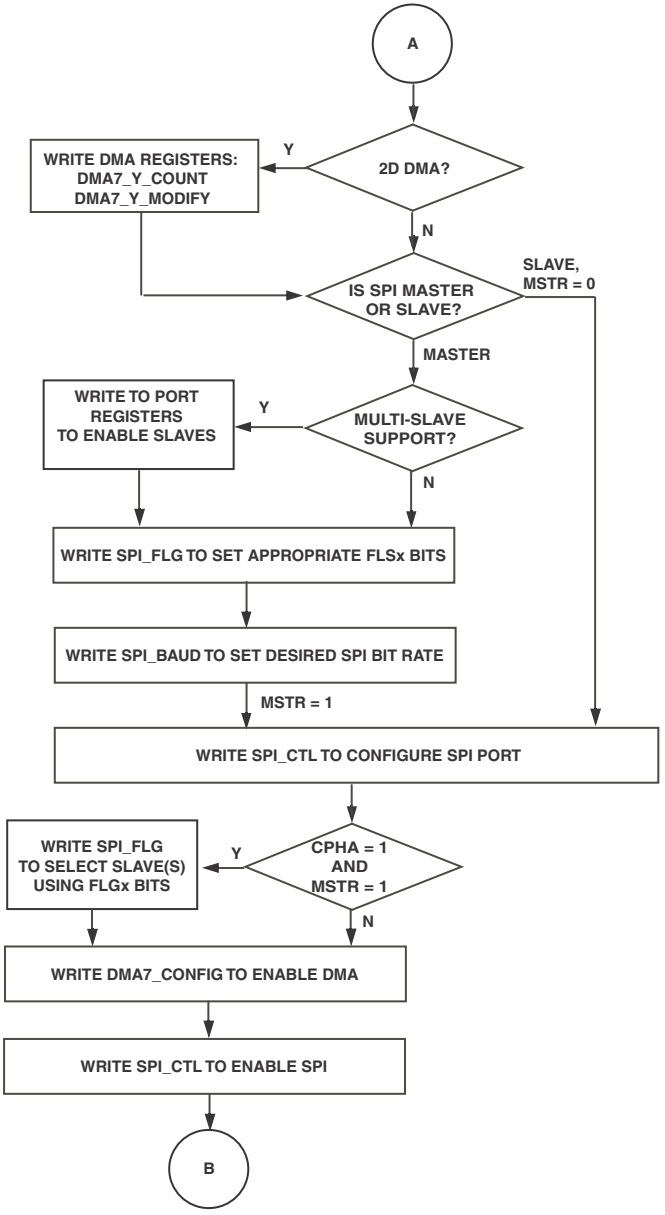


Figure 13-10. SPI DMA Flow Chart (Part 2 of 3)

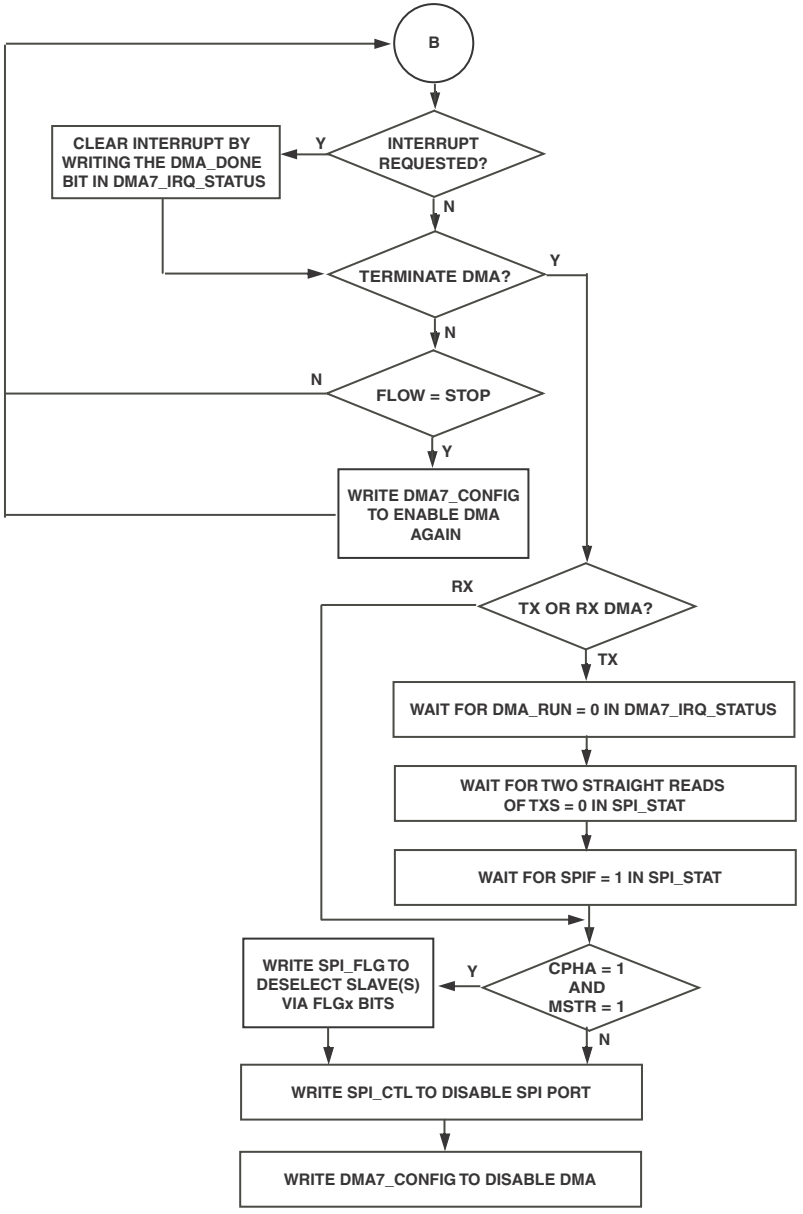


Figure 13-11. SPI DMA Flow Chart (Part 3 of 3)

SPI Registers

The SPI peripheral includes a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPI_BAUD, SPI_CTL, SPI_FLG, and SPI_STAT. Two registers are used for buffering receive and transmit data: SPI_RDBR and SPI_TDBR. The shift register, SFDR, is internal to the SPI module and is not directly accessible.

Table 13-3shows the functions of the SPI registers. Figure 13-12 through Figure 13-18 on page 13-43 provide details.

Table 13-3. SPI Register Mapping

Register Name	Function	Notes
SPI_BAUD	SPI port baud control	Value of “0” or “1” disables the serial clock
SPI_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPI_FLG	SPI port flag	Bits 0 and 8 are reserved
SPI_STAT	SPI port status	SPIF bit can be set by clearing SPE in SPI_CTL
SPI_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPI_CTL)
SPI_RDBR	SPI port receive data buffer	When register is read, hardware events can be triggered
SPI_SHADOW	SPI port data	Register has the same contents as SPI_RDBR, but no action is taken when it is read

SPI Baud Rate (SPI_BAUD) Register

The SPI_BAUD register is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

SCK frequency = (peripheral clock frequency SCLK)/(2 × SPI_BAUD)

Writing a value of “0” or “1” to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

Table 13-4 lists several possible baud rate values for SPI_BAUD.

Table 13-4. SPI Master Baud Rate Example

SPI_BAUD Decimal Value	SPI Clock (SCK) Divide Factor	Baud Rate for SCLK at 100 MHz
0	N/A	N/A
1	N/A	N/A
2	4	25 MHz
3	6	16.7 MHz
4	8	12.5 MHz
65,535 (0xFFFF)	131,070	763 Hz

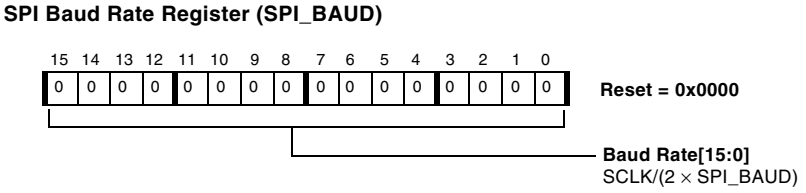


Figure 13-12. SPI Baud Rate Register

SPI Control (SPI_CTL) Register

The SPI_CTL register is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (SIZE) bit in SPI_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to b#00, a SPI port transaction is begun when the receive buffer is read. Data from the first read will need to be discarded since the read is needed to initiate the first SPI port transaction. When set to b#01, the transaction is initiated when the transmit buffer is written. A value of b#10 selects DMA receive mode and the first transaction is initiated by enabling the SPI for DMA receive mode. Subsequent individual transactions are initiated by a DMA read of the SPI_RDBR register. A value of 11 selects DMA transmit mode and the transaction is initiated by a DMA write of the SPI_TDBR register.

The PSSE bit is used to enable the SPISS input for an external master. When not used, SPISS can be disabled, freeing up a pin for an alternate function.

The EMISO bit enables the MISO pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the SPI_STAT register is set. See “Mode Fault Error (MODF)” on page 13-39.

Figure 13-13 on page 13-36 provides the bit descriptions for SPI_CTL.

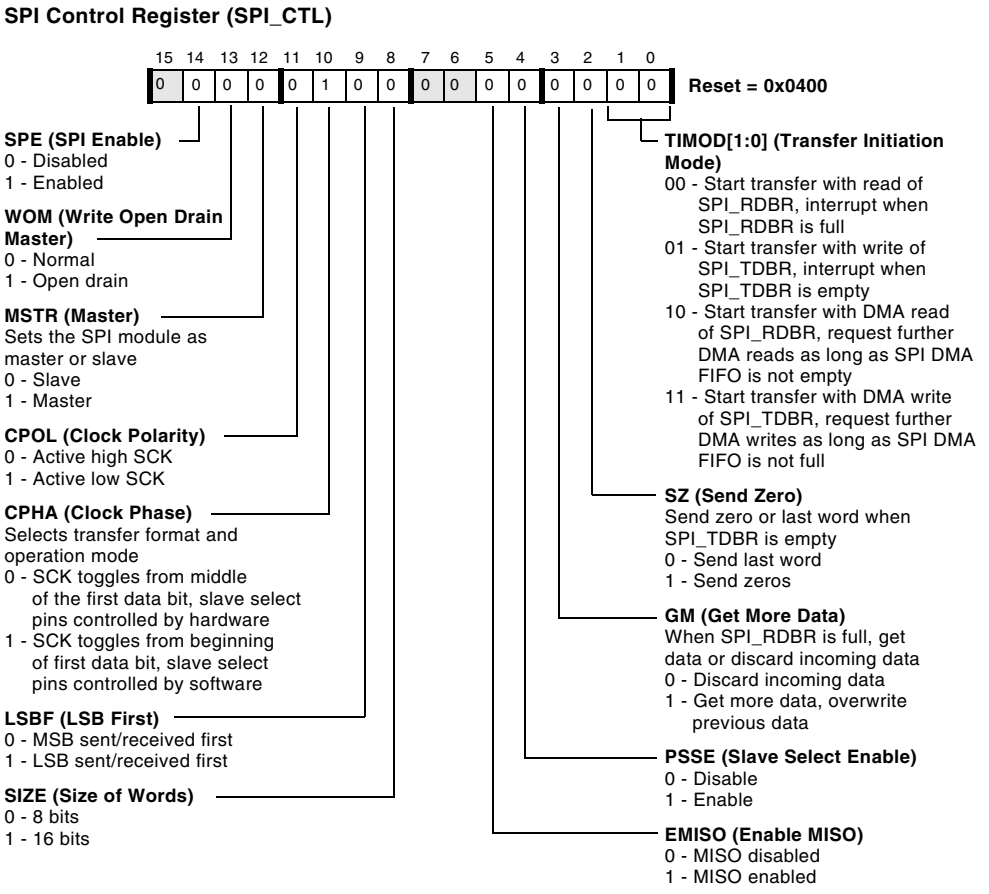

SPI Control Register (SPI_CTL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
Reset = 0x0400
SPE (SPI Enable)
0 - Disabled
1 - Enabled
WOM (Write Open Drain Master)
0 - Normal
1 - Open drain
MSTR (Master)
Sets the SPI module as master or slave
0 - Slave
1 - Master
CPOL (Clock Polarity)
0 - Active high SCK
1 - Active low SCK
CPHA (Clock Phase)
Selects transfer format and operation mode
0 - SCK toggles from middle of the first data bit, slave select pins controlled by hardware
1 - SCK toggles from beginning of first data bit, slave select pins controlled by software
LSBF (LSB First)
0 - MSB sent/received first
1 - LSB sent/received first
SIZE (Size of Words)
0 - 8 bits
1 - 16 bits
TIMOD[1:0] (Transfer Initiation Mode)
00 - Start transfer with read of SPI_RDBR, interrupt when SPI_RDBR is full
01 - Start transfer with write of SPI_TDBR, interrupt when SPI_TDBR is empty
10 - Start transfer with DMA read of SPI_RDBR, request further DMA reads as long as SPI DMA FIFO is not empty
11 - Start transfer with DMA write of SPI_TDBR, request further DMA writes as long as SPI DMA FIFO is not full
SZ (Send Zero)
Send zero or last word when SPI_TDBR is empty
0 - Send last word
1 - Send zeros
GM (Get More Data)
When SPI_RDBR is full, get data or discard incoming data
0 - Discard incoming data
1 - Get more data, overwrite previous data
PSSE (Slave Select Enable)
0 - Disable
1 - Enable
EMISO (Enable MISO)
0 - MISO disabled
1 - MISO enabled

Figure 13-13. SPI Control Register

SPI Flag (SPI_FLG) Register

The SPI_FLG register consists of two sets of bits that function as follows.

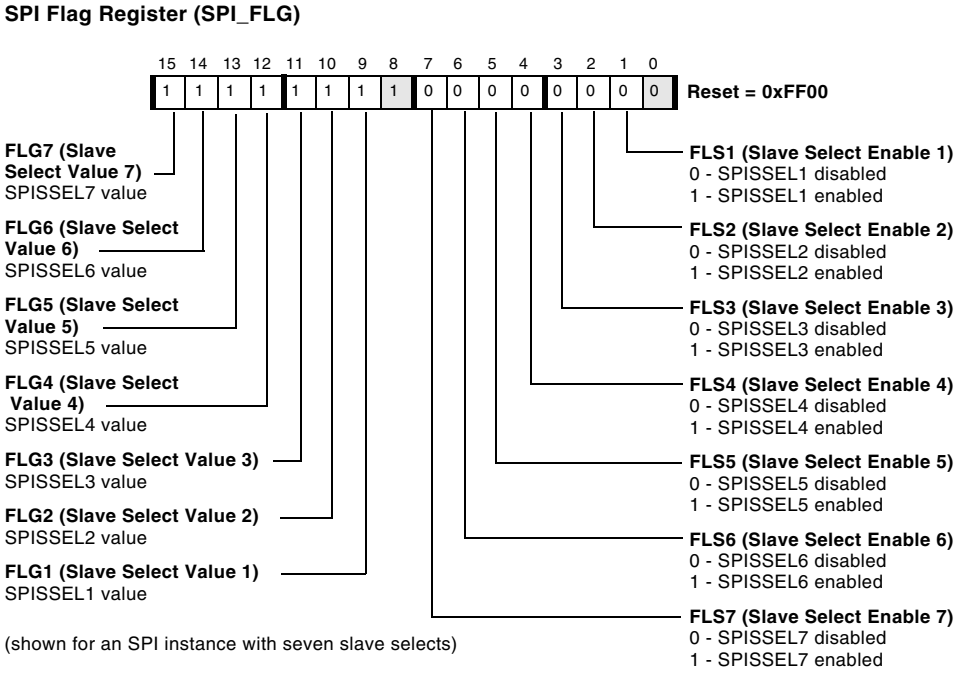

SPI Flag Register (SPI_FLG)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
Reset = 0xFF00
FLG7 (Slave Select Value 7)
SPISSEL7 value
FLG6 (Slave Select Value 6)
SPISSEL6 value
FLG5 (Slave Select Value 5)
SPISSEL5 value
FLG4 (Slave Select Value 4)
SPISSEL4 value
FLG3 (Slave Select Value 3)
SPISSEL3 value
FLG2 (Slave Select Value 2)
SPISSEL2 value
FLG1 (Slave Select Value 1)
SPISSEL1 value
(shown for an SPI instance with seven slave selects)
FLS1 (Slave Select Enable 1)
0 - SPISSEL1 disabled
1 - SPISSEL1 enabled
FLS2 (Slave Select Enable 2)
0 - SPISSEL2 disabled
1 - SPISSEL2 enabled
FLS3 (Slave Select Enable 3)
0 - SPISSEL3 disabled
1 - SPISSEL3 enabled
FLS4 (Slave Select Enable 4)
0 - SPISSEL4 disabled
1 - SPISSEL4 enabled
FLS5 (Slave Select Enable 5)
0 - SPISSEL5 disabled
1 - SPISSEL5 enabled
FLS6 (Slave Select Enable 6)
0 - SPISSEL6 disabled
1 - SPISSEL6 enabled
FLS7 (Slave Select Enable 7)
0 - SPISSEL7 disabled
1 - SPISSEL7 enabled

Figure 13-14. SPI Flag Register (example with 7 slave selects)

- Slave select enable (FLSx) bits

  Each FLSx bit corresponds to a general purpose port pin. When an FLSx bit is set, the corresponding port pin is driven as a slave select. For example, if FLS1 is set in SPI_FLG, the port pin corresponding to SPISSEL1 is driven as a slave select.

If the FLSx bit is not set, the general-purpose port registers configure and control the corresponding port pins.

- Slave select value (FLGx) bits
- When a port pin is configured as a slave select output, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPI_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, setting FLS3 in the SPI_FLG register drives the SPISSEL3 pin as a slave select. Then, clearing FLG3 in the SPI_FLG register drives the pin low, and setting FLG3 drives it high. The pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, the pin remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use the slave select function on a port pin to which it is mapped, it is only necessary to set the appropriate FLS bit in SPI_FLG. It is not necessary to write to an FLG bit, because the SPI hardware automatically drives the port pin.

SPI Status (SPI_STAT) Register

The SPI_STAT register, shown in Figure 13-15, is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPI_STAT register can be read at any time.

Some of the bits in SPI_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a “1” to the desired bit position of SPI_STAT. For example, if the TXE bit is set, the user must write

a “1” to bit 2 of SPI_STAT to clear the TXE error condition. This allows the user to read SPI_STAT without changing its value.

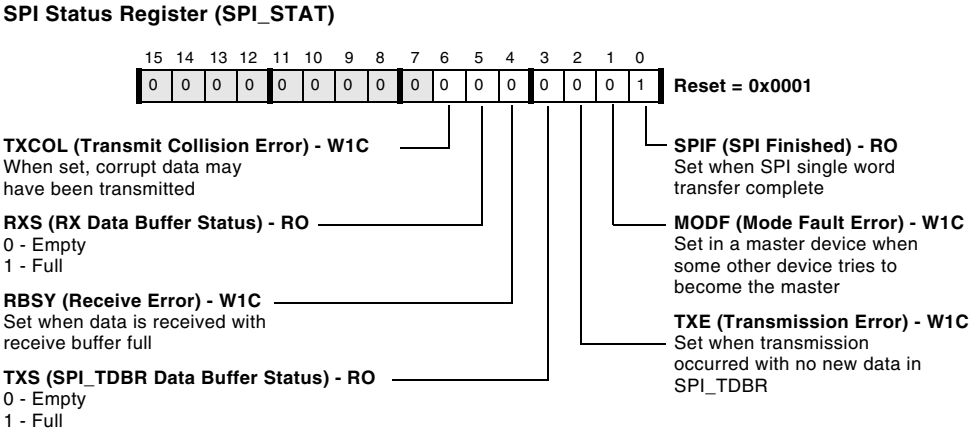

SPI Status Register (SPI_STAT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Reset = 0x0001
TXCOL (Transmit Collision Error) - W1C
When set, corrupt data may have been transmitted
RXS (RX Data Buffer Status) - RO
0 - Empty
1 - Full
RBSY (Receive Error) - W1C
Set when data is received with receive buffer full
TXS (SPI_TDBR Data Buffer Status) - RO
0 - Empty
1 - Full
SPIF (SPI Finished) - RO
Set when SPI single word transfer complete
MODF (Mode Fault Error) - W1C
Set in a master device when some other device tries to become the master
TXE (Transmission Error) - W1C
Set when transmission occurred with no new data in SPI_TDBR

Figure 13-15. SPI Status Register

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

Mode Fault Error (MODF)

The MODF bit is set in SPI_STAT when the SPISS input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPI_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:

- The MSTR control bit in SPI_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPI_CTL is cleared, disabling the SPI system

- The MODF status bit in SPI_STAT is set
- An SPI error interrupt is generated

These four conditions persist until the MODF bit is cleared by software. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either SPE or MSTR while MODF is set.

When MODF is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the SPISS input pin should be checked to make sure the pin is high. Otherwise, once SPE and MSTR are set, another mode fault error condition immediately occurs.

When SPE and MSTR are cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled. However, the slave select output pins revert to being controlled by the general-purpose I/O port registers. This could lead to contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an MODF error occurs, the program must configure the general-purpose I/O port registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the port pins that will be used as slave selects. Programs can do this by configuring the direction of the port pins prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave selects are automatically reconfigured as port pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The TXE bit is set in SPI_STAT when all the conditions of transmission are met, and there is no new data in SPI_TDBR (SPI_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPI_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPI_STAT register when a new transfer is completed, but before the previous data can be read from SPI_RDBR. The state of the GM bit in the SPI_CTL register determines whether SPI_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPI_STAT when a write to SPI_TDBR coincides with the load of the shift register. The write to SPI_TDBR can be by software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPI_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

SPI Transmit Data Buffer (SPI_TDBR) Register

The SPI_TDBR register is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPI_TDBR is loaded into the internal shift register SFDR. A read of SPI_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPI_TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPI_TDBR are repeatedly transmitted. A write to SPI_TDBR is permitted in this mode, and this data is transmitted.

If the SZ control bit in the SPI_CTL register is set, SPI_TDBR may be reset to zero under certain circumstances.

If multiple writes to SPI_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPI_TDBR are transmitted. Multiple writes to SPI_TDBR are possible, but not recommended.

SPI Transmit Data Buffer Register (SPI_TDBR)

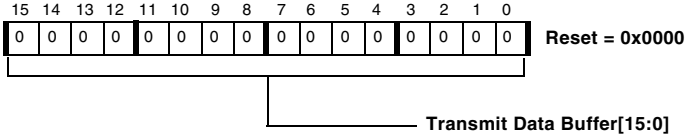


Figure 13-16. SPI Transmit Data Buffer Register

SPI Receive Data Buffer (SPI_RDBR) Register

The SPI_RDBR register is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPI_RDBR. During a DMA receive operation, the data in SPI_RDBR is automatically read by the DMA controller. When SPI_RDBR is read by software, the RXS bit in the SPI_STAT register is cleared and an SPI transfer may be initiated (if TIMOD = b#00).

SPI Receive Data Buffer Register (SPI_RDBR)
Read Only

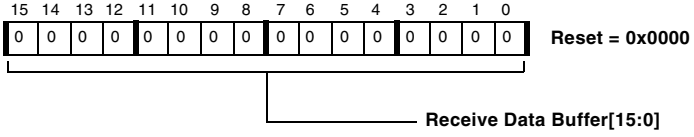


Figure 13-17. SPI Receive Data Buffer Register

SPI RDBR Shadow (SPI_SHADOW) Register

The SPI_SHADOW register is provided for use in debugging software. This register is at a different address than the receive data buffer, SPI_RDBR, but its contents are identical to that of SPI_RDBR. When a software read of SPI_RDBR occurs, the RXS bit in SPI_STAT is cleared and an SPI transfer may be initiated (if TIMOD = b#00 in SPI_CTL). No such hardware action occurs when the SPI_SHADOW register is read. The SPI_SHADOW register is read-only.

SPI RDBR Shadow Register (SPI_SHADOW)
Read Only

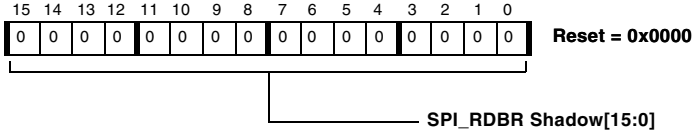


Figure 13-18. SPI RDBR Shadow Register

Programming Examples

This section includes examples (Listing 13-1 through Listing 13-8 on page 13-51) for both core-generated and DMA-based transfers. Each code example assumes that the appropriate processor header files are included.

Core-Generated Transfer

The following core-driven master-mode SPI example shows how to initialize the hardware, signal the start of a transfer, handle the interrupt and issue the next transfer, and generate a stop condition.

Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 13-1. SPI Register Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7);      /* FLS7 */
  W[P0] = R0;           /* Enable slave-select output pin */

  P0.H = hi(SPI_BAUD);
  P0.L = lo(SPI_BAUD);
  R0.L = 0x208E;        /* Write to SPI Baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133 MHz, SPI clock ~= 8 kHz
*/

/* Setup SPI Control Register */
/***********************************************
 * TIMOD [1:0] = 00 : Transfer On RDBR Read.
 * SZ [2]      = 0 : Send Last Word When TDBR Is Empty
 * GM [3]      = 1 : Overwrite Previous Data If RDBR Is Full
 * PSSE [4]    = 0 : Disables Slave-Select As Input (Master)
 * EMISO [5]   = 0 : MISO Disabled For Output (Master)
 * [7] and [6] = 0 : RESERVED
 * SIZE [8]    = 1 : 16 Bit Word Length Select
 * LSBF [9]    = 0 : Transmit MSB First
 * CPHA [10]   = 0 : Hardware Controls Slave-Select Outputs
 * CPOL [11]   = 1 : Active Low SCK
 * MSTR [12]   = 1 : Device Is Master
 * WOM [13]    = 0 : Normal MOSI/MISO Data Output (No Open Drain)
 * SPE [14]    = 1 : SPI Module Is Enabled
```

```
 * [15]        = 0 : RESERVED
 ***********************************************/
  P0.H = hi(SPI_CTL) ;
  P0.L = lo(SPI_CTL) ;
  R0 = 0x5908;
  W[P0] = R0.L; ssync;   /* Enable SPI as MASTER */
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following a dummy read of SPI_RDBR. Typically, known data which is desired to be transmitted to the slave is preloaded into the SPI_TDBR. In the following code, P1 is assumed to point to the start of the 16-bit transmit data buffer and P2 is assumed to point to the start of the 16-bit receive data buffer. In addition, the user must ensure appropriate interrupts are enabled for SPI operation.

Listing 13-2. Initiate Transfer

```
Initiate_Transfer:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITCLR (R0,0xF);      /* FLG7 */
  W[P0] = R0;           /* Drive 0 on enabled slave-select pin */

  P0.H = hi(SPI_TDBR);  /* SPI Transmit Register */
  P0.L = lo(SPI_TDBR);
  R0 = W[P1++] (z);
/* Get First Data To Be Transmitted And Increment Pointer */
  W[P0] = R0;            /* Write to SPI_TDBR */

  P0.H = hi(SPI_RDBR);
```

```
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z); /* Dummy read of SPI_RDBR kicks off transfer */
```

Post Transfer and Next Transfer

Following the transfer of data, the SPI generates an interrupt, which is serviced if the interrupt is enabled during initialization. In the interrupt routine, software must write the next value to be transmitted prior to reading the byte received. This is because a read of the SPI_RDBR initiates the next transfer.

Listing 13-3. SPI Interrupt Handler

```
SPI_Interrupt_Handler:
Process_SPI_Sample:
  P0.H = hi(SPI_TDBR);     /* SPI transmit register */
  P0.L = lo(SPI_TDBR);
  R0 = W[P1++](z);         /* Get next data to be transmitted */
  W[P0] = R0.l;            /* Write that data to SPI_TDBR */

Kick_Off_Next:
  P0.H = hi(SPI_RDBR);     /* SPI receive register */
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z);   /* Read SPI receive register (also kicks off
next transfer) */
  W[P2++] = R0;            /* Store received data to memory */
  RTI;                     /* Exit interrupt handler */
```

Stopping

In order for a data transfer to end after the user has transferred all data, the following code can be used to stop the SPI. Note that this is typically done in the interrupt handler to ensure the final data has been sent in its entirety.

Listing 13-4. Stopping SPI

```
Stopping_SPI:
  P0.H = hi(SPI_CTL);
 P0.L = lo(SPI_CTL);
  R0 = W[P0];
  BITCLR(R0, 14);         /* Clear SPI enable bit */
  W[P0] = R0.L; ssync;    /* Disable SPI */
```

DMA-Based Transfer

The following DMA-driven master-mode SPI autobuffer example shows how to initialize DMA, initialize SPI, signal the start of a transfer, and generate a stop condition.

DMA Initialization Sequence

The following code initializes the DMA to perform a 16-bit memory read DMA operation in autobuffer mode, and generates an interrupt request when the buffer has been sent. This code assumes that `P1` points to the start of the data buffer to be transmitted and that `NUM_SAMPLES` is a defined macro indicating the number of elements being sent.

Listing 13-5. DMA Initialization

```
Initialize_DMA:     /* Assume DMA7 as the channel for SPI DMA */
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R0 = 0x1084(z);   /* Autobuffer mode, IRQ on complete, linear
16-bit, mem read */
  w[P0] = R0;

  P0.H = hi(DMA7_START_ADDR);
  P0.L = lo(DMA7_START_ADDR);
```

```
[p0] = p1;     /* Start address of TX buffer */

P0.H  = hi(DMA7_X_COUNT);
P0.L  = lo(DMA7_X_COUNT);
R0    = NUM_SAMPLES;
w[p0] = R0;   /* Number of samples to transfer */
R0 = 2;
P0.H = hi(DMA7_X_MODIFY);
P0.L = lo(DMA7_X_MODIFY);
w[p0] = R0;   /* 2 byte stride for 16-bit words */

R0 = 1;       /* single dimension DMA means 1 row */
P0.H = hi(DMA7_Y_COUNT);
P0.L = lo(DMA7_Y_COUNT);
w[p0] = R0;
```

SPI Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 13-6. SPI Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7);    /* FLS7 */
  W[P0] = R0;         /* Enable slave-select output pin */

  P1.H = hi(SPI_BAUD);
  P1.L = lo(SPI_BAUD);
  R0.L = 0x208E;      /* Write to SPI baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133MHz, SPI clock ~= 8kHz */
```

```
  /* Setup SPI Control Register */
/************************************************
 * TIMOD [1:0] = 11 : Transfer on DMA TDBR write
 * SZ [2]      =  0 : Send last word when TDBR is empty
 * GM [3]      =  1 : Discard incoming data if RDBR is full
 * PSSE [4]    =  0 : Disables slave-select as input (master)
 * EMISO [5]   =  0 : MISO disabled for output (master)
 * [7] and [6] =  0 : RESERVED
 * SIZE [8]    =  1 : 16 Bit word length select
 * LSBF [9]    =  0 : Transmit MSB first
 * CPHA [10]   =  0 : SCK starts toggling at START of first data
bit
 * CPOL [11]   =  1 : Active HIGH serial clock
 * MSTR [12]   =  1 : Device is master
 * WOM [13]    =  0 : Normal MOSI/MISO data output (no open
drain)
 * SPE [14]    =  1 : SPI module is enabled
 * [15]        =  0 : RESERVED
 ************************************************/
  /* Configure SPI as MASTER */
  R1 = 0x190B(z);     /* Leave disabled until DMA is enabled */
  P1.L = lo(SPI_CTL);
  W[P1] = R1; ssync;
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following enabling of SPI. However, the DMA must be enabled before enabling the SPI.

Listing 13-7. Starting a Transfer

```
Initiate_Transfer:
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R2 = w[P0](z);
  BITSET (R2, 0);   /*Set DMA enable bit */
  w[p0] = R2.L;     /* Enable TX DMA */

  P4.H = hi(SPI_CTL);
  P4.L = lo(SPI_CTL);
  R2=w[p4](z);
  BITSET (R2, 14);  /* Set SPI enable bit */
  w[p4] = R2;       /* Enable SPI */
```

Stopping a Transfer

In order for a data transfer to end after the DMA has transferred all required data, the following code is executed in the SPI DMA interrupt handler. The example code below clears the DMA interrupt, then waits for the DMA engine to stop running. When the DMA engine has completed, `SPI_STAT` is polled to determine when the transmit buffer is empty. If there is data in the SPI Transmit FIFO, it is loaded as soon as the `TXS` bit clears. A second consecutive read with the `TXS` bit clear indicates the FIFO is empty and the last word is in the shift register. Finally, polling for the `SPIF` bit determines when the last bit of the last word has been shifted out. At that point, it is safe to shut down the SPI port and the DMA engine.

Listing 13-8. Stopping a Transfer

```
SPI_DMA_INTERRUPT_HANDLER:
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = 1 ;
  W[P0] = R0 ;  /* Clear DMA interrupt */

  /* Wait for DMA to complete */
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = DMA_RUN;      /* 0x08 */

CHECK_DMA_COMPLETE:  /* Poll for DMA_RUN bit to clear */
  R3 = W[P0] (Z);
  R1 = R3 & R0;
  CC = R1 == 0;
  IF !CC JUMP CHECK_DMA_COMPLETE;

  /* Wait for TXS to clear */
  P0.L = lo(SPI_STAT);
  P0.H = hi(SPI_STAT);
  R1 = TXS;   /* 0x08 */

Check_TXS:    /* Poll for TXS = 0 */
  R2 = W[P0] (Z);
  R2 = R2 & R1;
  CC = R0 == 0;
  IF !CC JUMP Check_TXS;

  R2 = W[P0] (Z);  /* Check if TXS stays clear for 2 reads */
  R2 = R2 & R1;
  CC = R0 == 0;
  IF !CC JUMP Check_TXS;
```

```
/* Wait for final word to transmit from SPI */
Final_Word:
  R0 = W[P0](Z);
  R2 = SPIF;   /* 0x01 */
  R0 = R0 & R2;
  CC = R0 == 0;
  IF CC JUMP Final_Word;

Disable_SPI:
  P0.L = lo(SPI_CTL);
  P0.H = hi(SPI_CTL);
  R0 = W[P0] (Z);
  BITCLR (R0,0xe);   /* Clear SPI enable bit */
  W[P0] = R0;        /* Disable SPI */

Disable_DMA:
  P0.L = lo(DMA7_CONFIG);
  P0.H = hi(DMA7_CONFIG);
  R0 = W[P0](Z);
  BITCLR (R0,0x0);   /* Clear DMA enable bit */
  W[P0] = R0;        /* Disable DMA */

  RTI;   /* Exit Handler */
```

Unique Information for the ADSP-BF59x Processor

None.

14 SPORT CONTROLLER

This chapter describes the synchronous serial peripheral port (SPORT). Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of SPORTs for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For SPORT DMA channel assignments, refer to Table 5-9 on page 5-103 in Chapter 5, "Direct Memory Access".

For SPORT interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, "System Interrupts".

To determine how each of the SPORTs is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, "General-Purpose Ports".

For a list of MMR addresses for each SPORT, refer to Chapter A, "System MMR Assignments".

SPORT behavior for the ADSP-BF59x that differs from the general information in this chapter can be found at the end of this chapter in the section "Unique Information for the ADSP-BF59x Processor" on page 14-73.

Overview

Unlike the SPI interface which has been designed for SPI-compatible communication only, the SPORT modules support a variety of serial data communication protocols, for example:

- A-law or µ-law companding according to G.711 specification
- Multichannel or time-division-multiplexed (TDM) modes
- Stereo audio I^2S mode
- H.100 telephony standard support

In addition to these standard protocols, the SPORT module provides modes to connect to standard peripheral devices, such as ADCs or codecs, without external glue logic. With support for high data rates, independent transmit and receive channels, and dual data paths, the SPORT interface is a perfect choice for direct serial interconnection between two or more processors in a multiprocessor system. Many processors provide compatible interfaces, including processors from Analog Devices and other manufacturers.

Each SPORT has its own set of control registers and data buffers.

Features

A SPORT can operate at up to ½ the system clock (`SCLK`) rate for an internally generated or external serial clock. The SPORT external clock must always be less than the `SCLK` frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

A SPORT offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.
- Provides alternate framing and control for interfacing to I^2S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
- Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT.
- Provides two synchronous transmit and two synchronous receive data signals and buffers to double the total supported datastreams.
- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 14-28 for more information.)
- Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source.
- Operates with or without frame synchronization signals for each data word, with internally generated or externally generated frame signals, with active high or active low frame signals, and with either of two configurable pulse widths and frame signal timing.
- Performs interrupt-driven, single word transfers to and from on-chip memory under processor control.

- Provides direct memory access transfer to and from memory under DMA master control. DMA can be autobuffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).
- Has a multichannel mode for TDM interfaces. A SPORT can receive and transmit data selectively from a time-division-multiplexed serial bitstream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 (= 1023 – 128). Note the multichannel select registers and the WSIZE register control which subset of the 128 channels within the active region can be accessed.

Interface Overview

A SPORT provides an I/O interface to a wide variety of peripheral serial devices. SPORTs provide synchronous serial data transfer only. Each SPORT has one group of signals (primary data, secondary data, clock, and frame sync) for transmit and a second set of signals for receive. The receive and transmit functions are programmed separately. A SPORT is a full duplex device, capable of simultaneous data transfer in both directions. A SPORT can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

Figure 14-1 on page 14-6 shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT_TX register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX shift register. The bits in the shift register are shifted out on the DTPRI/DTSEC pin, MSB first or LSB first, synchronous to the serial clock on the TSCLK pin. The receive portion of the SPORT accepts data from the DRPRI/DRSEC pin synchronous to the serial clock on the RSCLK pin. When an entire word

is received, the data is optionally expanded, then automatically transferred to the SPORT_RX register, and then into the RX FIFO where it is available to the processor. Table 14-1 shows the signals for each SPORT.

Table 14-1. SPORT Signals

Pin	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary
RSCLKx	Receive Clock
RFSx	Receive Frame Sync

A SPORT receives serial data on its DRPRI and DRSEC inputs and transmits serial data on its DTPRI and DTSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTPRI and DTSEC) are synchronous to the transmit clock (TSCLK). For receive, the data bits (DRPRI and DRSEC) are synchronous to the receive clock (RSCLK). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFS and TFS are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins, if enabled by a specific processor port configuration, provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary signals is interleaved in main memory and can be retrieved by setting a stride in the data address generators (DAG) unit. For more information about

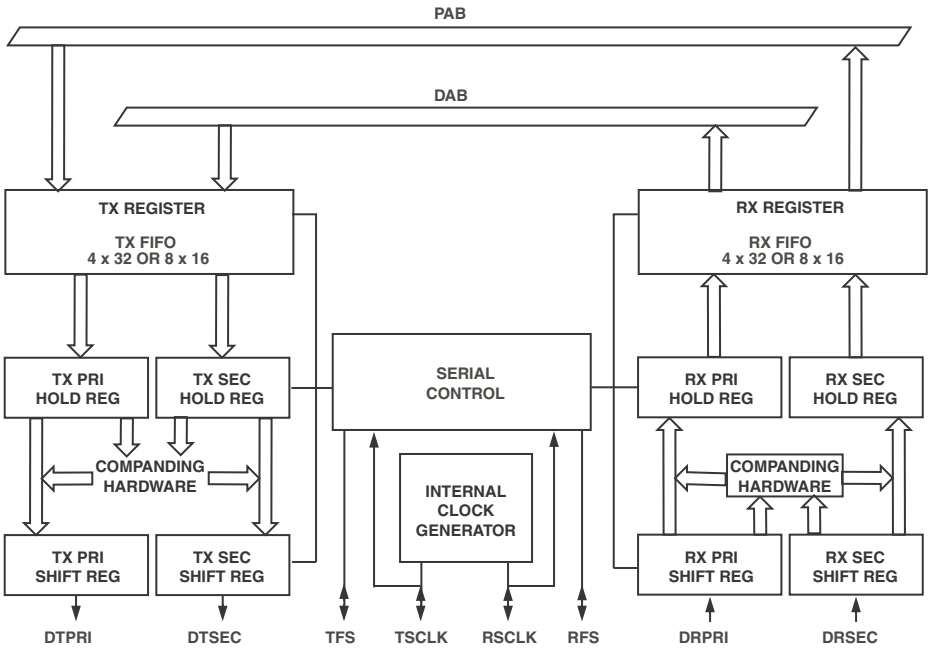


Figure 14-1. SPORT Block Diagram[1, 2, 3]

1 All wide arrow data paths are 16- or 32-bits wide, depending on SLEN. for SLEN = 2 to 15, a 16-bit data path with 8-deep fifo is used. for SLEN = 16 to 31, a 32-bit data path with 4-deep fifo is used.
2 TX register is the bottom of the TX fifo, RX register is the top of the RX fifo.
3 In multichannel mode, the TFS pin acts as transmit data valid (TDV). For more information, see "Multichannel Operation" on page 14-14.

DAGs, see the *Data Address Generators* chapter in the *Blackfin Processor Programming Reference*. Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor's powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

Figure 14-2 shows a possible port connection for a device with at least two SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices 1, 2, …N.

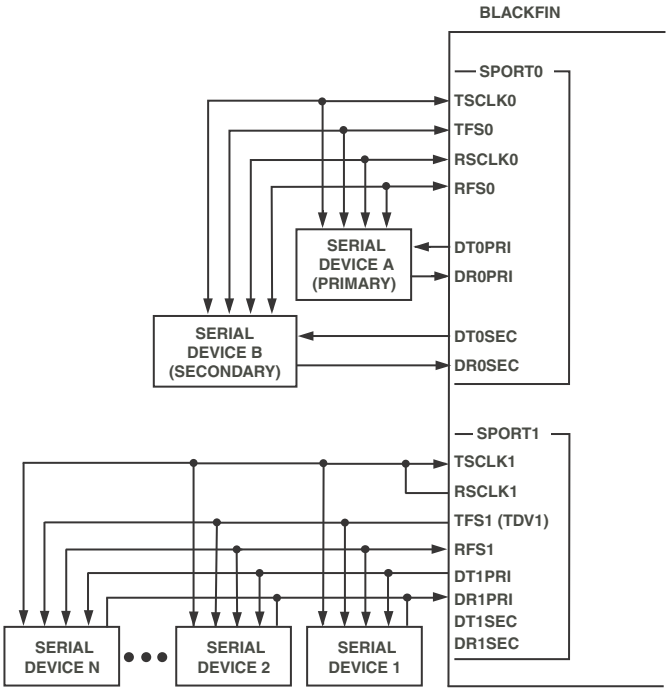


Figure 14-2. Example SPORT Connections (SPORT0 is Standard Mode, SPORT1 is Multichannel Mode)[1, 2]

1 In multichannel mode, TFS functions as a transmit data valid (TDV) output. See "Multichannel Operation" on page 14-14.

2 Although shown as an external connection, the TSCLK1/RSCLK1 connection is internal in multichannel mode. See "Multichannel Operation" on page 14-14.

Figure 14-3 shows an example of a stereo serial device with three transmit and two receive channels connected to a processor with two SPORTs.

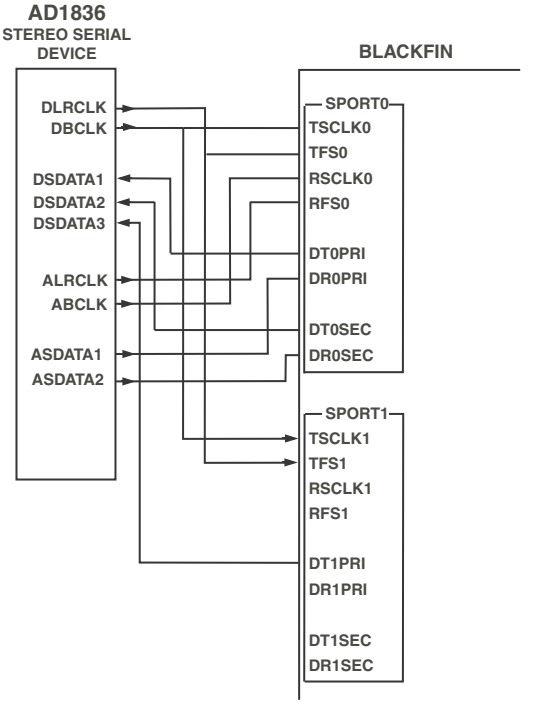


Figure 14-3. Stereo Serial Connection

SPORT Pin/Line Terminations

The processor has very fast drivers on all output pins, including the SPORTs. If connections on the data, clock, or frame sync lines are longer than six inches, consider using a series termination for strip lines on point-to-point connections. This may be necessary even when using low speed serial clocks, because of the edge rates.

Description of Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT_TX register readies the SPORT for transmission. The TFS signal initiates the transmission of serial data. Once transmission has begun, each value written to the SPORT_TX register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORT_TCR1 register. Each bit is shifted out on the driving edge of TSCLK. The driving edge of TSCLK can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed.

SPORT Disable

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT's transmit or receive enable bits (TSPEN in the SPORT_TCR1 register and RSPEN in the SPORT_RCR1 register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the SPORT_TCR1, SPORT_TCR2, SPORT_RCR1, and SPORT_RCR2 registers (including the TSPEN and RSPEN enable bits) and the SPORT_TCLKDIV, SPORT_RCLKDIV, SPORT_TFSDIVx, and SPORT_RFSDIVx clock and frame sync divisor registers. Any ongoing operations are aborted.

Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFS/RFS and TSCLK/RSCLK outputs, if these signals are configured to be generated internally.

When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. A SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORT_RCLKDIV, SPORT_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORT_TCR1 and/or RSPEN in SPORT_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in "SPORT Registers" on page 14-44. All control and status bits in the SPORT registers are active high unless otherwise noted.

Stereo Serial Operation

Several stereo serial modes can be supported by the SPORT, including the popular I2S format. To use these modes, set bits in the SPORT_RCR2 or SPORT_TCR2 registers. Setting RSFSE or TSFSE in SPORT_RCR2 or SPORT_TCR2 changes the operation of the frame sync pin to a left/right

clock as required for I2S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 14-4 on page 14-13 and Figure 14-5 on page 14-14 show timing diagrams for stereo serial mode operation.

Table 14-2 on page 14-11 shows several modes that can be configured using bits in SPORT_TCR1 and SPORT_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user's needs, without changing the standard, is indicated by an "X."

Blackfin SPORTs are designed such that, in I^2S master mode, LRCLK is held at the last driven logic level and does not transition, to provide an edge, after the final data word is driven out. Therefore, while transmitting a fixed number of words to an I^2S receiver that expects an LRCLK edge to receive the incoming data word, the SPORT should send a dummy word after transmitting the fixed number of words. The transmission of this dummy word toggles LRCLK, generating an edge. Transmission of the dummy word is not required when the I^2S receiver is a Blackfin SPORT.

Table 14-2. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
RSFSE	1	1	0
RRFST	0	0	0
LARFS	0	1	0
LRFS	0	1	0
RFSR	1	1	1
RCKFE	1	0	0

Table 14-2. Stereo Serial Settings (Continued)

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
SLEN	2 – 31	2 – 31	2 – 31
RLSBIT	0	0	0
RFSDIV (If internal FS is selected.)	2 – Max	2 – Max	2 – Max
RXSE (Secondary Enable is available for RX and TX.)	X	X	X

Note most bits shown as a 0 or 1 may be changed depending on the user's preference, creating many other "almost standard" modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 14-2 on page 14-11 provide glueless interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I^2S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 14-25) still apply, but now refer to one half the period and twice the frequency. For instance, setting RFSDIV or TFSDIV = 31 produces an LRCLK that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The LRFS bit determines the polarity of the RFS or TFS frame sync pin for the channel that is considered a "right" channel. Thus, setting LRFS = 0 (meaning that it is an active high signal) indicates that the frame sync is high for the "right" channel, thus implying that it is low for the "left" channel. This is the default setting.

The RRFST and TRFST bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary DRSEC and DTSEC pins are useful extensions of the SPORT which pair well with stereo serial mode. Multiple I^{2}S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (frame sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 14-3 on page 14-8, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.

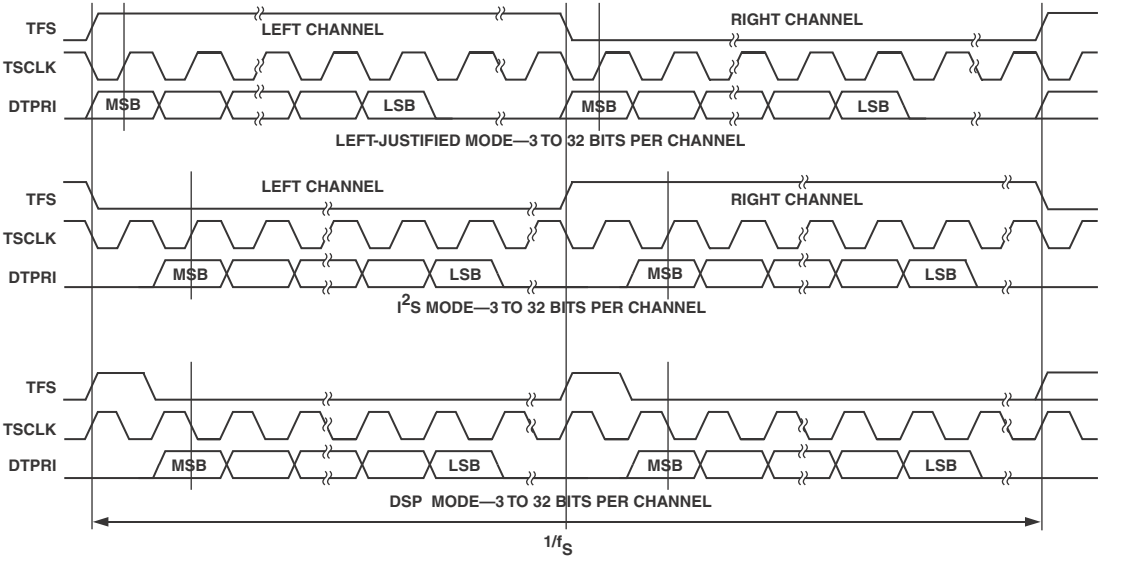


Figure 14-4. SPORT Stereo Serial Modes, Transmit[1, 2, 3]

1 DSP mode does not identify channel.
2 TFS normally operates at f_S except for DSP mode which is 2 x f_S.
3 TSCLK frequency is normally 64 x TFS but may be operated in burst mode.

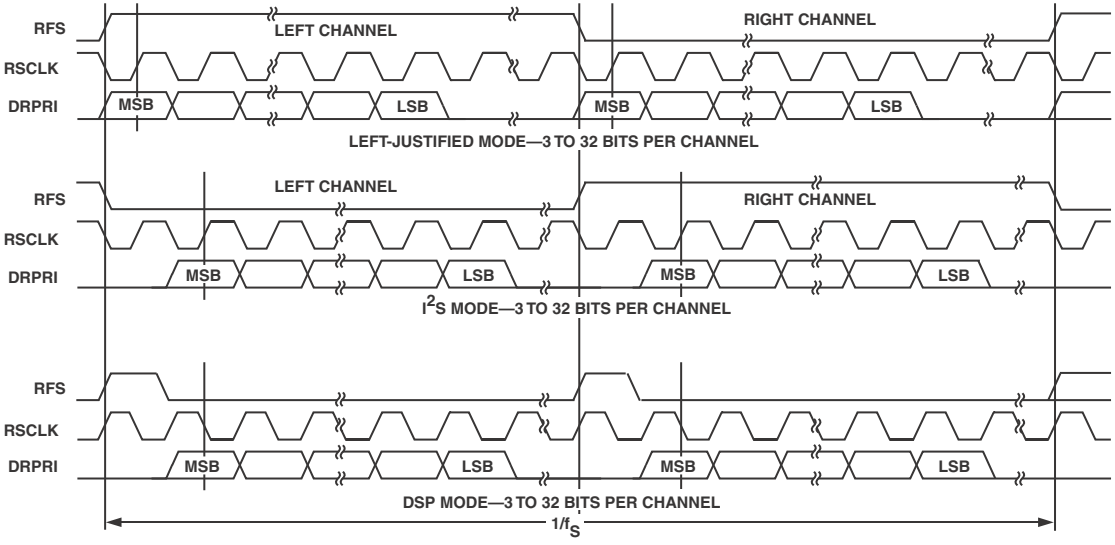


Figure 14-5. SPORT Stereo Serial Modes, Receive[1, 2, 3]

1 DSP mode does not identify channel.
2 RFS normally operates at f_S except for DSP mode which is 2 x f_S.
3 RSCLK frequency is normally 64 × RFS but may be operated in burst mode.

Multichannel Operation

The SPORT offers a multichannel mode of operation which allows the SPORT to communicate in a time-division-multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bitstream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024 total channels. RX and TX must use the same 128-channel region to

selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORT_TCR1 register), unless it is in multichannel mode and an inactive time slot occurs. The DTSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORT_TCR2 register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORT_TX or SPORT_RX operation for multichannel mode. This is especially important in "DMA data unpacked mode," since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORT_MCMC2) must be enabled prior to enabling SPORT_TX or SPORT_RX operation. When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

Figure 14-6 on page 14-16 shows example timing for a multichannel transfer that has these characteristics:

- Use TDM method where serial data is sent or received on different channels sharing the same serial bus
- Can independently select transmit and receive channels
- RFS signals start of frame
- TFS is used as "transmit data valid" for external logic, true only during transmit channels
- Receive on channels 0 and 2, transmit on channels 1 and 2
- Multichannel frame delay is set to 1

See "Timing Examples" on page 14-38 for more examples.

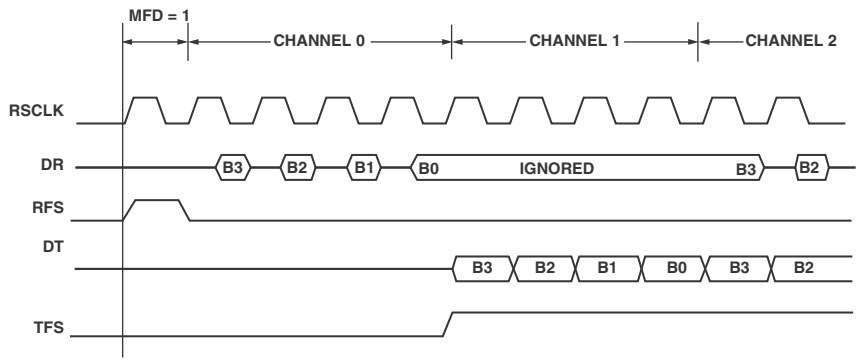


Figure 14-6. Multichannel Operation

Multichannel Enable

Setting the MCMEN bit in the SPORT_MCM2 register enables multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

Setting the MCMEN bit enables multichannel operation for *both* the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.

When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.

Table 14-3 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 14-3. Multichannel Mode Configuration

SPORT_RCR1 or SPORT_RCR2	SPORT_TCR1 or SPORT_TCR2	Notes
RSPEN	TSPEN	Set or clear both
IRCLK	-	Independent
-	ITCLK	Independent
RDTYPE	TDTYPE	Independent
RLSBIT	TLSBIT	Independent
IRFS	-	Independent
-	ITFS	Ignored
RFSR	TFSR	Ignored
-	DITFS	Ignored
LRFS	LTFS	Independent
LARFS	LATFS	Both must be 0
RCKFE	TCKFE	Set or clear both to same value

Table 14-3. Multichannel Mode Configuration (Continued)

SPORT_RCR1 or SPORT_RCR2	SPORT_TCR1 or SPORT_TCR2	Notes
SLEN	SLEN	Set or clear both to same value
RXSE	TXSE	Independent
RSFSE	TSFSE	Both must be 0
RRFST	TRFST	Ignored

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFS signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFS as a frame sync. This is true whether RFS is generated internally or externally. The RFS signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFS indicates the beginning of the channel 0 data word.

Since RFS is used by both the SPORT_TX and SPORT_RX channels of the SPORT in multichannel mode configuration, the corresponding bit pairs in SPORT_RCR1 and SPORT_TCR1, and in SPORT_RCR2 and SPORT_TCR2, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if SPORT_RX operation is not enabled.

In multichannel mode, RFS timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFS signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFS signal serves as an

output-enabled signal for the data transmit pin. The SPORT drives TFS in multichannel mode whether or not ITFS is cleared. The TFS pin in multichannel mode still obeys the LTFS bit. If LTFS is set, the transmit data valid signal will be active low—a low signal on the TFS pin indicates an active channel.

Once the initial RFS is received, and a frame transfer has started, all other RFS signals are ignored by the SPORT until the complete frame has been transferred.

If MFD > 0, the RFS may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In multichannel mode, the RFS signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFS signals required. Therefore, internally generated frame syncs are always data independent.

The Multichannel Frame

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of 1 – 1024 channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the multichannel select registers. See Figure 14-7 on page 14-20.

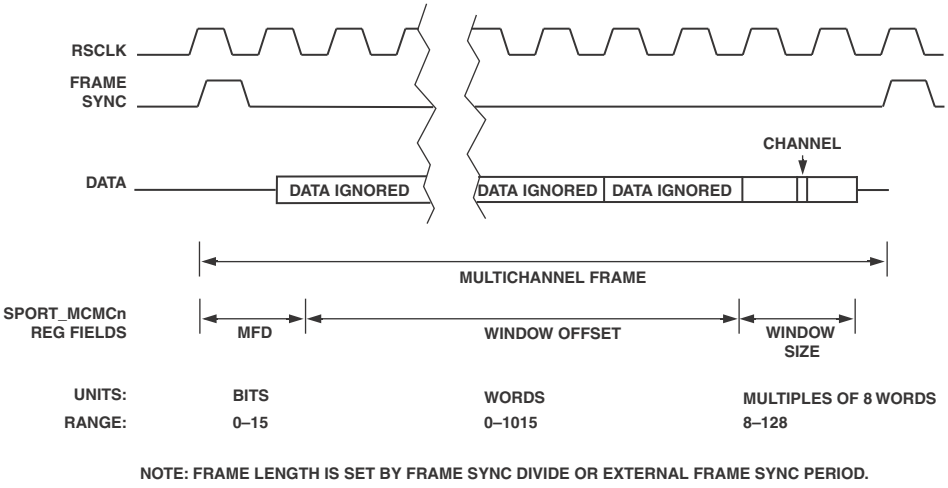


Figure 14-7. Relationships for Multichannel Parameters

Multichannel Frame Delay

The 4-bit MFD field in SPORT_MCMC2 specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of MFD is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for MFD causes the frame sync to be concurrent with the first data bit. The maximum value allowed for MFD is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size (WSIZE[3:0]) defines the number of channels that can be enabled/disabled by the multichannel select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum active

window size of 8 channels. To calculate the active window size from the WSIZE register, use this equation:

Number of words in active window = 8 × (WSIZE + 1)

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Window Offset

The window offset (WOFF[9:0]) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 (WSIZE = 0) and an offset of 93 (WOFF = 93). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

Other Multichannel Fields in SPORT_MCMC2

The FSDR bit in the SPORT_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (When FSDR = 0), the data is transmitted on the same edge that the TFS is generated. For example, a positive edge on TFS causes data to be transmitted on the positive edge of the TSCLK—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a multibit word from 3 to 32 bits in length that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select which words are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 contiguous channels may be selected out of 1024 available channels. The SPORT_MRCSn and SPORT_MTCSn multichannel select registers are used to enable and disable individual channels; the SPORT_MRCSn registers specify the active receive channels, and the SPORT_MTCSn registers specify the active transmit channels.

Four registers make up each multichannel select register. Each of the four registers has 32 bits, corresponding to 32 channels. Setting a bit enables that channel, so the SPORT selects its word from the multiple word block of data (for either receive or transmit). See Figure 14-8.

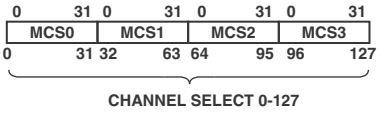


Figure 14-8. Multichannel Select Registers

Channel select bit 0 always corresponds to the first word of the active window. To determine a channel's absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in MCS2 selects word 71 of the active window to be enabled. Setting bit 2 in MCS1 selects word 34 of the active window, and so on.

Setting a particular bit in the SPORT_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. Clearing the bit in the SPORT_MTCSn register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORT_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the SPORT_RX buffer. Clearing the bit in the SPORT_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or μ-law companding is selected with the TDTYPE field in the SPORT_TCR1 register and the RDTYPE field in the SPORT_RCR1 register, and applies to all active channels. (See "Companding" on page 14-28 for more information about companding.)

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORT_MCMC2 multichannel configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguration is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words (unless the secondary side is enabled). The data to be

transmitted or received would be placed at addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the channel register to make sure that the active window is not being serviced. If the channel count is 0, then the multichannel select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

- Set for external frame sync. Frame sync generated by external bus master.
- `TFSR/RFSR` set (frame syncs required)
- `LTFS/LRFS` set (active low frame syncs)
- Set for external clock
- `MCMEN` set (multichannel mode selected)
- `MFD` = 0 (no frame delay between frame sync and first data bit)
- `SLEN` = 7 (8-bit words)
- `FSDR` = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2× Clock Recovery Control

The SPORT can recover the data rate clock from a provided 2× input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with the proper phase relationship. A 2-bit mode signal (`MCCRM[1:0]` in the

SPORT_MCMC2 register) chooses the applicable clock mode, which includes a non-divide or bypass mode for normal operation. A value of MCCRM = 00 chooses non-divide or bypass mode (H.100-compatible), MCCRM = 10 chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and MCCRM = 11 chooses HMVIP clock divide (extract 8 MHz from 16 MHz).

Functional Description

The following sections provide a functional description of the SPORT.

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is SCLK/2. The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the 16-bit serial clock divide modulus registers, SPORT_TCLKDIV and SPORT_RCLKDIV.

TSCLK frequency = (SCLK frequency)/(2 × (SPORT_TCLKDIV + 1))

RSCLK frequency = (SCLK frequency)/(2 × (SPORT_RCLKDIV + 1))

If the value of SPORT_TCLKDIV or SPORT_RCLKDIV is changed while the internal serial clock is enabled, the change in TSCLK or RSCLK frequency takes effect at the start of the drive edge of TSCLK or RSCLK that follows the next leading edge of TFS or RFS.

When an internal frame sync is selected (ITFS = 1 in the SPORT_TCR1 register or IRFS = 1 in the SPORT_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORT_TCLKDIV or SPORT_RCLKDIV has changed. The second frame sync will cause the update.

The SPORT_TFSDIV and SPORT_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFS or

RFS pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

# of transmit serial clocks between frame sync assertions = TFSDIV + 1

# of receive serial clocks between frame sync assertions = RFSDIV + 1

Use the following equations to determine the correct value of TFSDIV or RFSDIV, given the serial clock frequency and desired frame sync frequency:

SPORT TFS frequency = (TSCLK frequency)/(SPORT_TFSDIV + 1)

SPORT RFS frequency = (RSCLK frequency)/(SPORT_RFSDIV + 1)

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORT_TCR2 or SPORT_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late transmit frame syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See the *ADSP-BF592 Blackfin Processor Data Sheet* for exact timing specifications.

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORT_TCR2 and SPORT_RCR2 registers of each SPORT determines the word length according to this formula:

Serial Word Length = SLEN + 1

The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so SLEN≥ 3).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORT_RCR1 and SPORT_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORT_TCR1 register and the RDTYPE field of the SPORT_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 14-4 on page 14-28.

These formats are applied to serial data words loaded into the SPORT_RX and SPORT_TX buffers. SPORT_TX data words are not actually zero filled or sign extended, because only the significant bits are transmitted.

Table 14-4. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORT_TCR1 Data Formatting	SPORT_RCR1 Data Formatting
00	Normal operation	Zero fill
01	Reserved	Sign extend
10	Compand using μ-law	Compand using μ-law
11	Compand using A-law	Compand using A-law

Companding

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The SPORT supports the two most widely used companding algorithms, μ-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the SPORT_RX register is the right-justified, expanded value of the eight LSBs received and sign extended to 16 bits. A write to SPORT_TX causes the 16-bit value to be compressed to eight LSBs (sign extended to the width of the transmit word) and written to the internal transmit register. Although the companding standards support only 13-bit (A-law) or 14-bit (μ-law) maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

Clock Signal Options

Each SPORT has a transmit clock signal (TSCLK) and a receive clock signal (RSCLK). The clock signals are configured by the TCKFE and RCKFE bits of the SPORT_TCR1 and SPORT_RCR1 registers. Serial clock frequency is configured in the SPORT_TCLKDIV and SPORT_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORT_TCR1 configuration register and the IRCLK bit in the SPORT_RCR1 configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the processor, and the TSCLK or RSCLK pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORT_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLK or RSCLK pins, and the serial clock divisors in the SPORT_TCLKDIV/SPORT_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the system clock or with each other. The system clock must have a higher frequency than RSCLK and TSCLK.

When the SPORT uses external clocks, it must be enabled for a minimal number of stable clock pulses before the first active frame sync is sampled. Failure to allow for these clocks may result in a SPORT malfunction. See the *ADSP-BF592 Blackfin Processor Data Sheet* for details.

The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (transmit frame sync) and RFS (receive frame sync). A variety of framing options are available; these options are configured in the SPORT configuration registers (SPORT_TCR1, SPORT_TCR2, SPORT_RCR1 and SPORT_RCR2). The TFS and RFS signals of a SPORT are independent and are separately configured in the control registers.

Framed Versus Unframed

The use of multiple frame sync signals is optional in SPORT communications. The TFSR (transmit frame sync required select) and RFSR (receive frame sync required select) control bits determine whether frame sync signals are required. These bits are located in the SPORT_TCR1 and SPORT_RCR1 registers.

When TFSR = 1 or RFSR = 1, a frame sync signal is required for every data word. To allow continuous transmitting by the SPORT, each new data word must be loaded into the SPORT_TX hold register before the previous word is shifted out and transmitted.

When TFSR = 0 or RFSR = 0, the corresponding frame sync signal is not required. A single frame sync is needed to initiate communications but is ignored after the first bit is transferred. Data words are then transferred continuously, unframed.

With frame syncs not required, interrupt or DMA requests may not be serviced frequently enough to guarantee continuous unframed data flow. Monitor status bits or check for a SPORT Error interrupt to detect underflow or overflow of data.

Figure 14-9 on page 14-31 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORT_TCR1 and SPORT_RCR1 registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers.

See "Timing Examples" on page 14-38 for more timing examples.

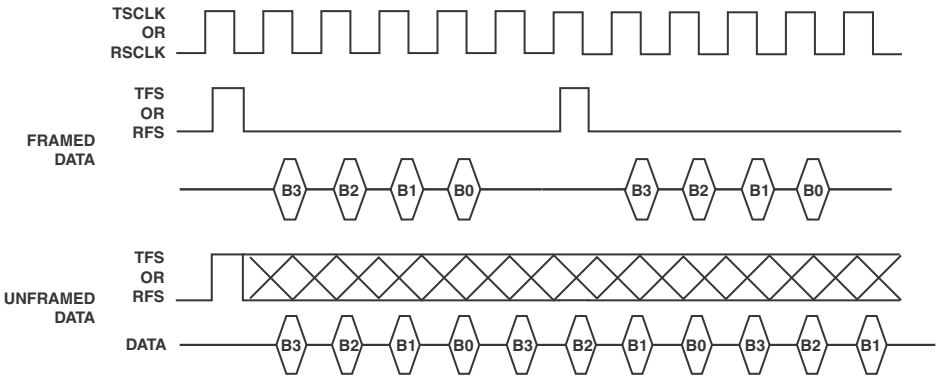


Figure 14-9. Framed Versus Unframed Data

Internal Versus External Frame Syncs

Both transmit and receive frame syncs can be independently generated internally or can be input from an external source. The ITFS and IRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers determine the frame sync source.

When ITFS = 1 or IRFS = 1, the corresponding frame sync signal is generated internally by the SPORT, and the TFS pin or RFS pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the SPORT_TFSDIV or SPORT_RFSDIV register.

When ITFS = 0 or IRFS = 0, the corresponding frame sync signal is accepted as an input on the TFS pin or RFS pin, and the frame sync divisors in the SPORT_TFSDIV/SPORT_RFSDIV registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be either active high or active low (in other words, inverted). The LTFS and LRFS bits of the SPORT_TCR1 and SPORT_RCR1 registers determine frame sync logic levels:

- When LTFS = 0 or LRFS = 0, the corresponding frame sync signal is active high.
- When LTFS = 1 or LRFS = 1, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The LTFS and LRFS bits are initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The TCKFE and RCKFE bits of the SPORT_TCR1 and SPORT_RCR1 registers select the driving and sampling edges of the serial data and frame syncs.

For the SPORT transmitter, setting TCKFE = 1 in the SPORT_TCR1 register selects the falling edge of TSCLK to drive data and internally generated frame syncs and selects the rising edge of TSCLK to sample externally gener-

ated frame syncs. Setting TCKFE = 0 selects the rising edge of TSCLK to drive data and internally generated frame syncs and selects the falling edge of TSCLK to sample externally generated frame syncs.

For the SPORT receiver, setting RCKFE = 1 in the SPORT_RCR1 register selects the falling edge of RSCLK to drive internally generated frame syncs and selects the rising edge of RSCLK to sample data and externally generated frame syncs. Setting RCKFE = 0 selects the rising edge of RSCLK to drive internally generated frame syncs and selects the falling edge of RSCLK to sample data and externally generated frame syncs.

Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (TCKFE = 1 in the SPORT_TCR1 register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for TCKFE in the transmitter and RCKFE in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 14-10, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

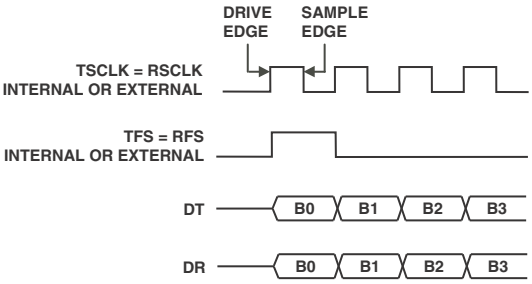


Figure 14-10. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 14-11 on page 14-34, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

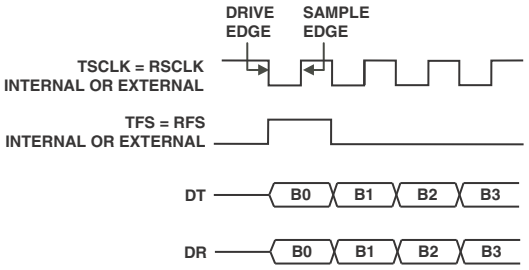


Figure 14-11. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

Early Versus Late Frame Syncs (Normal Versus Alternate Timing)

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The LATFS and LARFS bits of the SPORT_TCR1 and SPORT_RCR1 registers configure this option.

When LATFS = 0 or LARFS = 0, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early framing mode. Continuous operation is restricted to word sizes of 4 or longer (SLEN ≥ 3).

When LATFS = 1 or LARFS = 1, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 14-12 on page 14-36 illustrates the two modes of frame signal timing. In summary:

- For the LATFS or LARFS bits of the SPORT_TCR1 or SPORT_RCR1 registers: LATFS = 0 or LARFS = 0 for early frame syncs, LATFS = 1 or LARFS = 1 for late frame syncs.
- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
- Data is transmitted MSB first (TLSBIT = 0 or RLSBIT = 0) or LSB first (TLSBIT = 1 or RLSBIT = 1).
- Frame sync and clock are generated internally or externally.

See "Timing Examples" on page 14-38 for more examples.

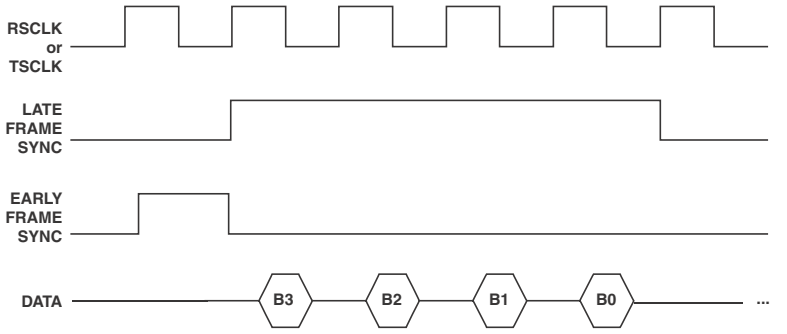


Figure 14-12. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFS) is output only when the SPORT_TX buffer has data ready to transmit. The data-independent transmit frame sync select bit (DITFS) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPORT_TCR1 register configures this option.

When DITFS = 0, the internally generated TFS is only output when a new data word has been loaded into the SPORT_TX buffer. The next TFS is generated once data is loaded into SPORT_TX. This mode of operation allows data to be transmitted only when it is available.

When DITFS = 1, the internally generated TFS is output at its programmed interval regardless of whether new data is available in the SPORT_TX buffer. Whatever data is present in SPORT_TX is transmitted again with each assertion of TFS. The TUVF (transmit underflow status) bit in the SPORT_STAT register is set when this occurs and old data is retransmitted. The TUVF status bit is also set if the SPORT_TX buffer does not have new data when an externally generated TFS occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFS is used, a single write to the SPORT_TX data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.

Peripheral Bus Errors

The SPORT generates a peripheral bus error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORT_TX)
- Writing a read-only register (for example, SPORT_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed" on page 14-30, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 14-34, and "Frame Syncs in Multichannel Mode" on page 14-18). This section contains additional examples to illustrate other possible combinations of the framing options.

These timing examples show the relationships between the signals but are not scaled to show the actual timing parameters of the processor. Consult the *ADSP-BF592 Blackfin Processor Data Sheet* for actual timing parameters and values.

These examples assume a word length of four bits (SLEN = 3). Framing signals are active high (LRFS = 0 and LTFS = 0).

Figure 14-13 on page 14-39 through Figure 14-18 on page 14-41 show framing for receiving data.

In Figure 14-13 and Figure 14-14, the normal framing mode is shown for non-continuous data (any number of TSCLK or RSCLK cycles between words) and continuous data (no TSCLK or SCLK cycles between words).

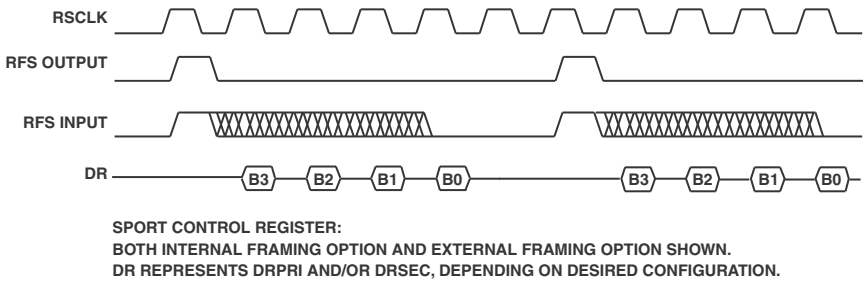


Figure 14-13. SPORT Receive, Normal Framing

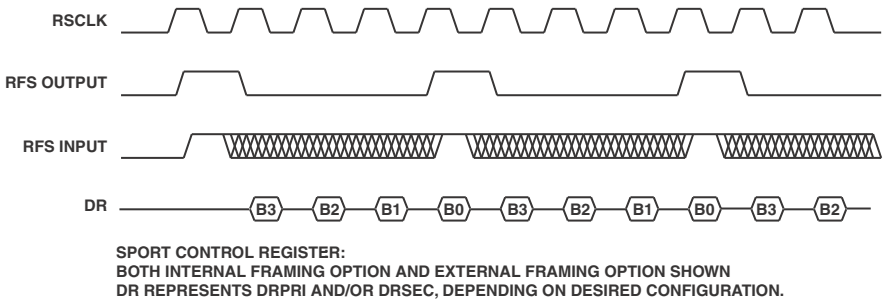


Figure 14-14. SPORT Continuous Receive, Normal Framing

Figure 14-15 on page 14-40 and Figure 14-16 on page 14-40 show non-continuous and continuous receiving in the alternate framing mode. These four figures show the input timing requirement for an externally generated frame sync and also the output timing characteristic of an internally generated frame sync. Note the output meets the input timing requirement; therefore, with two SPORT channels used, one SPORT channel could provide RFS for the other SPORT channel.

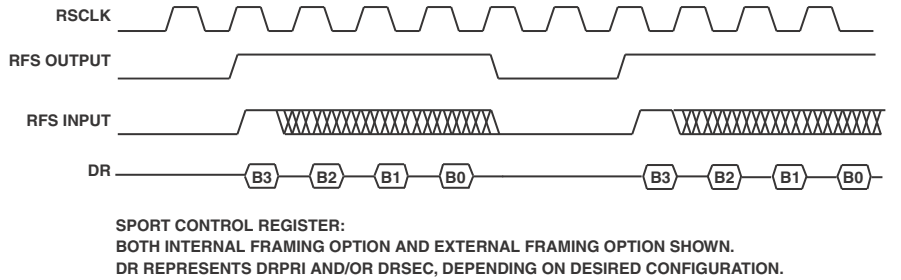


Figure 14-15. SPORT Receive, Alternate Framing

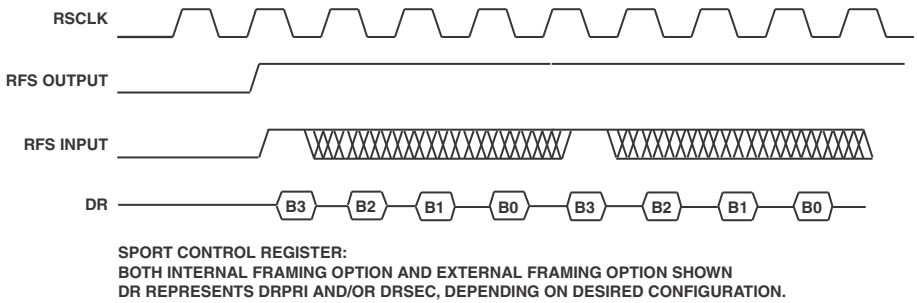


Figure 14-16. SPORT Continuous Receive, Alternate Framing

Figure 14-17 on page 14-41 and Figure 14-18 on page 14-41 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one RSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multiword bursts (continuous reception).

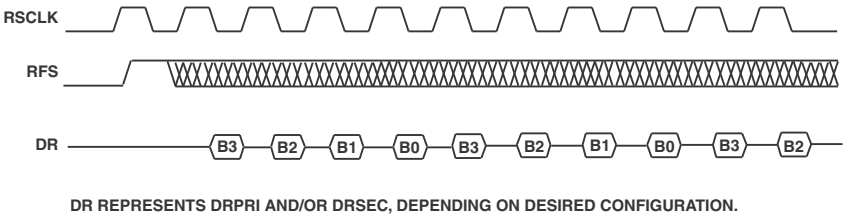


Figure 14-17. SPORT Receive, Unframed Mode, Normal Framing

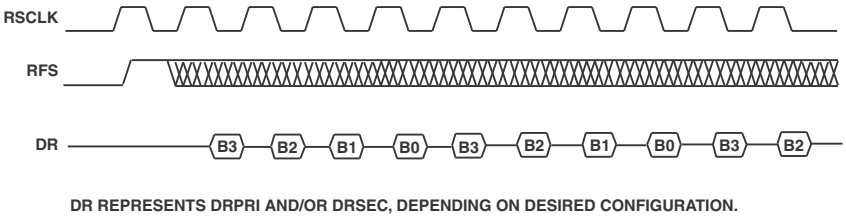


Figure 14-18. SPORT Receive, Unframed Mode, Alternate Framing

Figure 14-19 on page 14-42 through Figure 14-24 on page 14-43 show framing for transmitting data and are very similar to Figure 14-13 on page 14-39 through Figure 14-18 on page 14-41.

In Figure 14-19 on page 14-42 and Figure 14-20 on page 14-42, the normal framing mode is shown for non-continuous data (any number of TSCLK cycles between words) and continuous data (no TSCLK cycles between words). Figure 14-21 on page 14-42 and Figure 14-22 on page 14-43 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFS output meets the RFS input timing requirement.

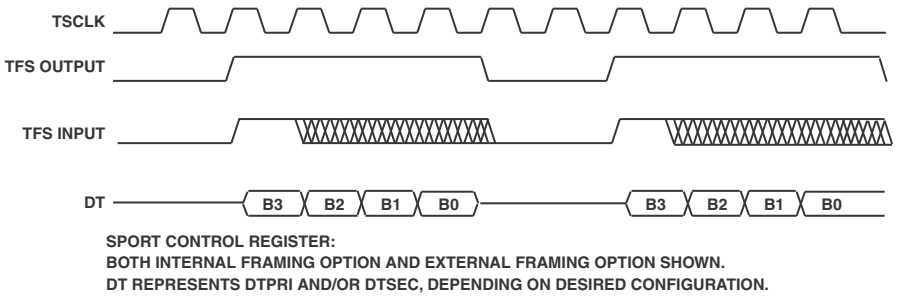


Figure 14-19. SPORT Transmit, Normal Framing

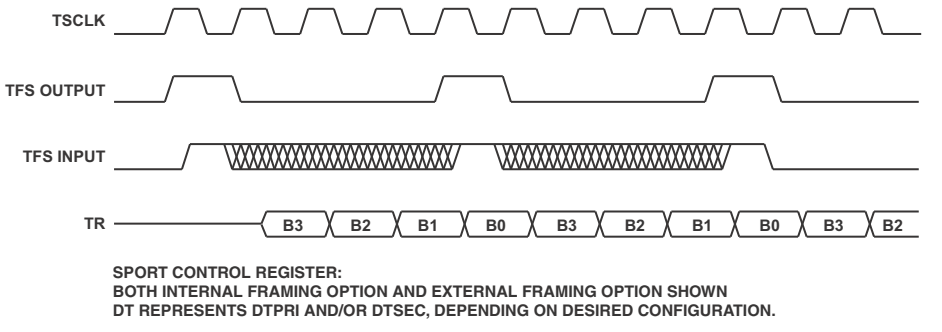


Figure 14-20. SPORT Continuous Transmit, Normal Framing

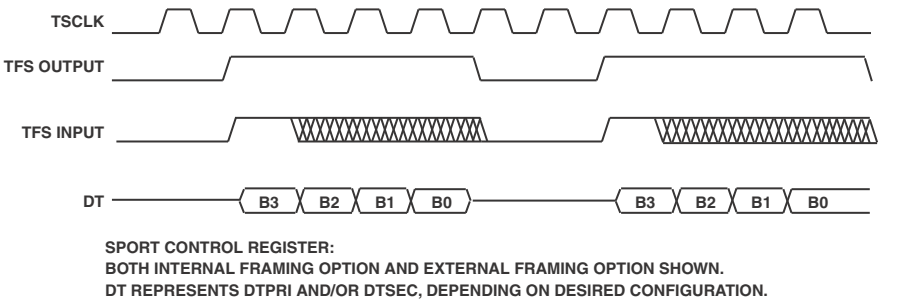


Figure 14-21. SPORT Transmit, Alternate Framing

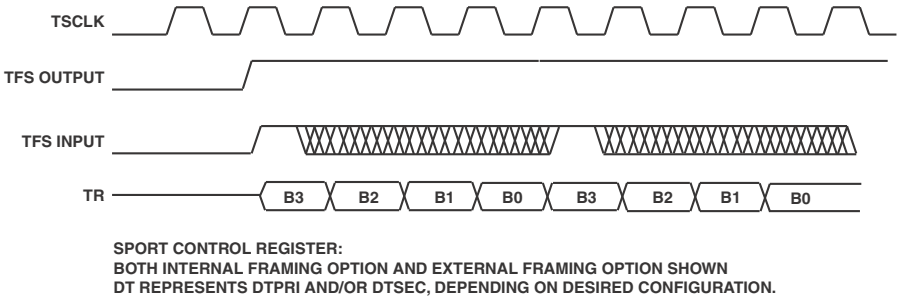


Figure 14-22. SPORT Continuous Transmit, Alternate Framing

Figure 14-23 on page 14-43 and Figure 14-24 on page 14-43 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

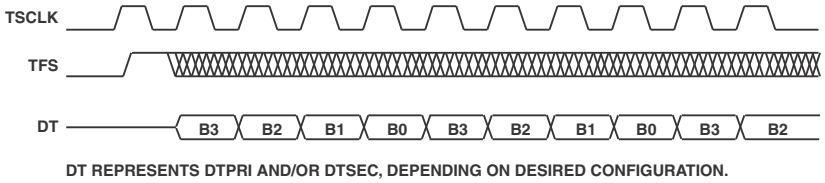


Figure 14-23. SPORT Transmit, Unframed Mode, Normal Framing

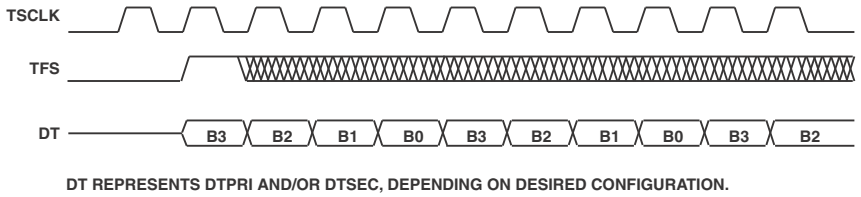


Figure 14-24. SPORT Transmit, Unframed Mode, Alternate Framing

SPORT Registers

The following sections describe the SPORT registers. Table 14-5 provides an overview of the available control registers.

Table 14-5. SPORT Register Mapping

Register Name	Function	Notes
SPORT_TCR1	Primary transmit configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORT_TCR2	Secondary transmit configuration register	
SPORT_TCLKDIV	Transmit clock divider register	Ignored if external SPORT clock mode is selected
SPORT_TFSDIV	Transmit frame sync divider register	Ignored if external frame sync mode is selected
SPORT_TX	Transmit data register	See description of FIFO buffering at "SPORT Transmit Data (SPORT_TX) Register" on page 14-56
SPORT_RCR1	Primary receive configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORT_RCR2	Secondary receive configuration register	
SPORT_RCLK_DIV	Receive clock divider register	Ignored if external SPORT clock mode is selected
SPORT_RFSDIV	Receive frame sync divider register	Ignored if external frame sync mode is selected
SPORT_RX	Receive data register	See description of FIFO buffering at "SPORT Receive Data (SPORT_RX) Register" on page 14-58
SPORT_STAT	Receive and transmit status	
SPORT_MCM1	Primary multichannel mode configuration register	Configure this register before enabling the SPORT

Table 14-5. SPORT Register Mapping (Continued)

Register Name	Function	Notes
SPORT_MCM2	Secondary multichannel mode configuration register	Configure this register before enabling the SPORT
SPORT_MRCSn	Receive channel selection registers	Select or deselect channels in a multichannel frame
SPORT_MTCSn	Transmit channel selection registers	Select or deselect channels in a multichannel frame
SPORT_CHNL	Currently serviced channel in a multichannel frame	

Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORT_RCLKDIV, SPORT_TCLKDIV, and multichannel mode channel select registers). The SPORT_TX register writes are always enabled; SPORT_RX, SPORT_CHNL, and SPORT_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and multichannel select registers.

SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORT_TCR1 and SPORT_TCR2, shown in Figure 14-25 on page 14-47 and Figure 14-26 on page 14-48.

A SPORT is enabled for transmit if bit 0 (TSPEN) of the transmit configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORT_TCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORT_TCR1 is not written except for bit 0 (TSPEN). For example,

```
write (SPORT_TCR1, 0x0001) ;   /* SPORT TX Enabled */
write (SPORT_TCR1, 0xFF01) ;   /* ignored, no effect */
write (SPORT_TCR1, 0xFFF0) ;   /* SPORT disabled, SPORT_TCR1
                                  still equal to 0x0000 */
```

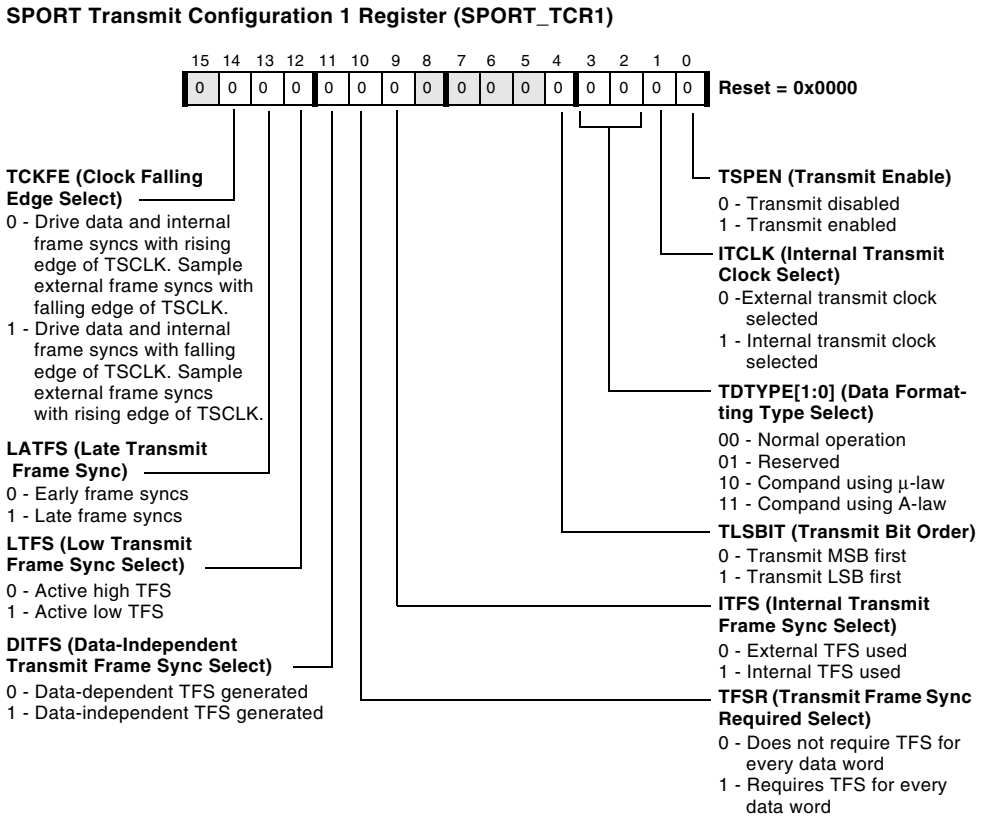

SPORT Transmit Configuration 1 Register (SPORT_TCR1)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
TCKFE (Clock Falling Edge Select)
0 - Drive data and internal frame syncs with rising edge of TSCLK. Sample external frame syncs with falling edge of TSCLK.
1 - Drive data and internal frame syncs with falling edge of TSCLK. Sample external frame syncs with rising edge of TSCLK.
LATFS (Late Transmit Frame Sync)
0 - Early frame syncs
1 - Late frame syncs
LTFS (Low Transmit Frame Sync Select)
0 - Active high TFS
1 - Active low TFS
DITFS (Data-Independent Transmit Frame Sync Select)
0 - Data-dependent TFS generated
1 - Data-independent TFS generated
TSPEN (Transmit Enable)
0 - Transmit disabled
1 - Transmit enabled
ITCLK (Internal Transmit Clock Select)
0 -External transmit clock selected
1 - Internal transmit clock selected
TDTYPE[1:0] (Data Formatting Type Select)
00 - Normal operation
01 - Reserved
10 - Compand using μ-law
11 - Compand using A-law
TLSBIT (Transmit Bit Order)
0 - Transmit MSB first
1 - Transmit LSB first
ITFS (Internal Transmit Frame Sync Select)
0 - External TFS used
1 - Internal TFS used
TFSR (Transmit Frame Sync Required Select)
0 - Does not require TFS for every data word
1 - Requires TFS for every data word

Figure 14-25. SPORT Transmit Configuration 1 Register

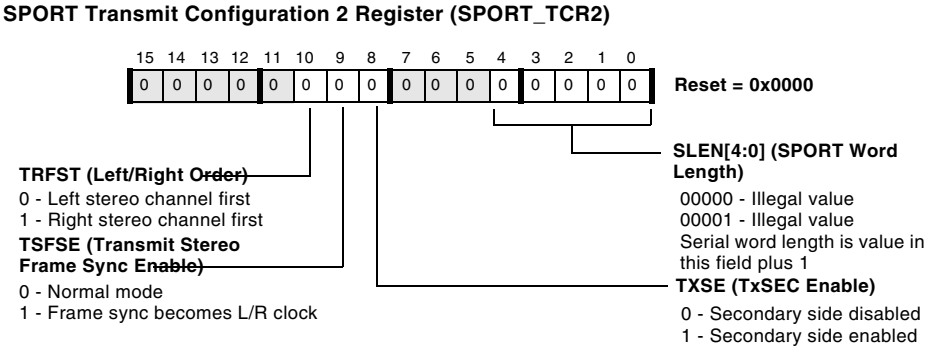

SPORT Transmit Configuration 2 Register (SPORT_TCR2)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SLEN[4:0] (SPORT Word Length)
00000 - Illegal value
00001 - Illegal value
Serial word length is value in this field plus 1
TRFST (Left/Right Order)
0 - Left stereo channel first
1 - Right stereo channel first
TSFSE (Transmit Stereo Frame Sync Enable)
0 - Normal mode
1 - Frame sync becomes L/R clock
TXSE (TxSEC Enable)
0 - Secondary side disabled
1 - Secondary side enabled

Figure 14-26. SPORT Transmit Configuration 2 Register

Additional information for the SPORT_TCR1 and SPORT_TCR2 transmit configuration register bits includes:

- **Transmit enable** (TSPEN). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting TSPEN causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting TSPEN.

  Similarly, if DMA transfers are used, DMA control should be configured correctly before setting TSPEN. Set all DMA control registers before setting TSPEN.

  Clearing TSPEN causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing TSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORT_TCR1 with all of the necessary bits, including TSPEN.

- **Internal transmit clock select**. (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. The two TDTYPE bits specify data formats used for single and multichannel operation.
- **Bit order select**. (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

  Serial Word Length = SLEN + 1;

  The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The SPORT always transfers the SLEN+1 lower bits from the transmit buffer.

The frame sync signal is controlled by the SPORT_TFSDIV and SPORT_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider

must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal transmit frame sync select.** (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).
- **Transmit frame sync required select.** (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a transmit frame sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent transmit frame sync select.** (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORT_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

  The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORT_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORT_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the SPORT_TX register on time. If the receiver can tolerate occasional late frame sync pulses, DITFS should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the SPORT_TX register.

- **Low transmit frame sync select.** (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).

- **Late transmit frame sync**. (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (TCKFE). This bit selects which edge of the TCLKx signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.
- **TxSec enable**. (TXSE). This bit enables the transmit secondary side of the SPORT (if set).
- **Stereo serial enable**. (TSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (TRFST). If this bit is set, the right channel is transmitted first in stereo serial operating mode. By default this bit is cleared, and the left channel is transmitted first.

SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers

The main control registers for the receive portion of each SPORT are the receive configuration registers, SPORT_RCR1 and SPORT_RCR2, shown in Figure 14-27 on page 14-52 and Figure 14-28 on page 14-53.

A SPORT is enabled for receive if bit 0 (RSPEN) of the receive configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

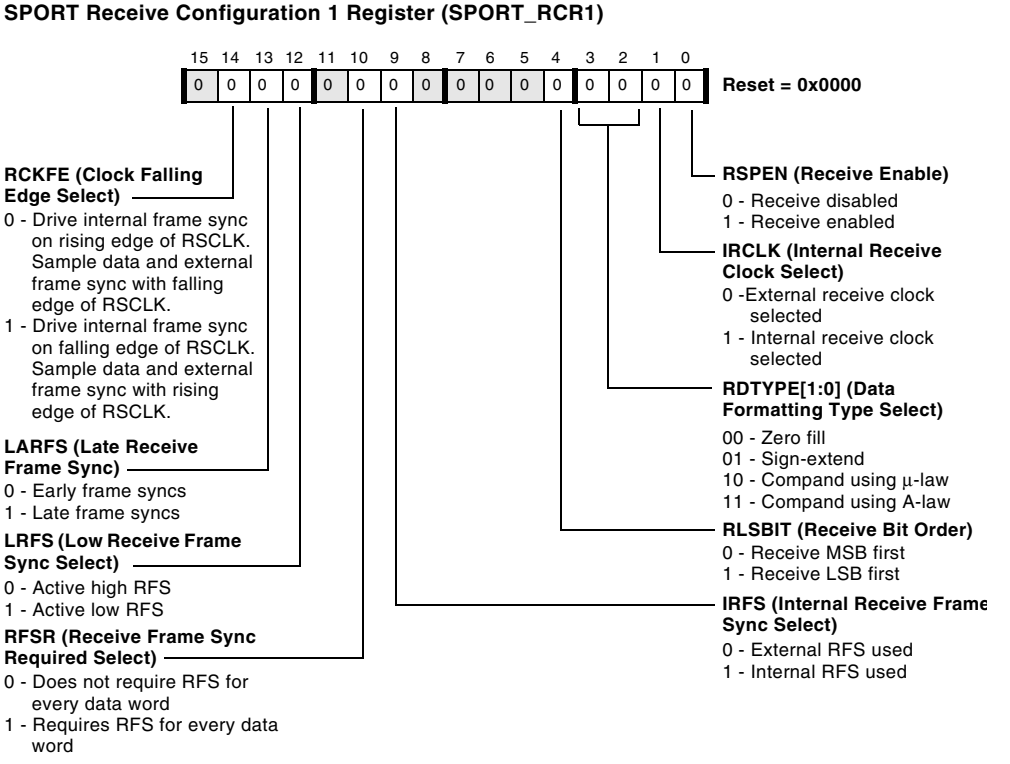

SPORT Receive Configuration 1 Register (SPORT_RCR1)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RCKFE (Clock Falling Edge Select)
0 - Drive internal frame sync on rising edge of RSCLK. Sample data and external frame sync with falling edge of RSCLK.
1 - Drive internal frame sync on falling edge of RSCLK. Sample data and external frame sync with rising edge of RSCLK.
LARFS (Late Receive Frame Sync)
0 - Early frame syncs
1 - Late frame syncs
LRFS (Low Receive Frame Sync Select)
0 - Active high RFS
1 - Active low RFS
RFSR (Receive Frame Sync Required Select)
0 - Does not require RFS for every data word
1 - Requires RFS for every data word
RSPEN (Receive Enable)
0 - Receive disabled
1 - Receive enabled
IRCLK (Internal Receive Clock Select)
0 -External receive clock selected
1 - Internal receive clock selected
RDTYPE[1:0] (Data Formatting Type Select)
00 - Zero fill
01 - Sign-extend
10 - Compand using µ-law
11 - Compand using A-law
RLSBIT (Receive Bit Order)
0 - Receive MSB first
1 - Receive LSB first
IRFS (Internal Receive Frame Sync Select)
0 - External RFS used
1 - Internal RFS used

Figure 14-27. SPORT Receive Configuration 1 Register

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORT_RCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORT_RCR1 is not written except for bit 0 (RSPEN). For example,

```
write (SPORT_RCR1, 0x0001) ;  /* SPORT RX Enabled */
write (SPORT_RCR1, 0xFF01) ;  /* ignored, no effect */
write (SPORT_RCR1, 0xFFF0) ;  /* SPORT disabled, SPORT_RCR1
                                 still equal to 0x0000 */
```

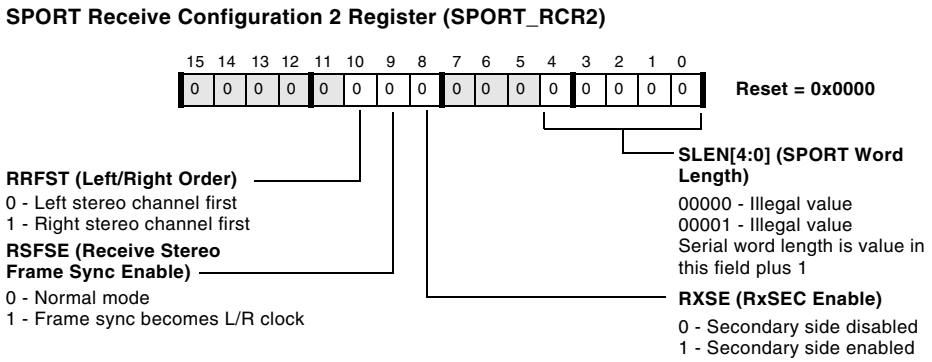

SPORT Receive Configuration 2 Register (SPORT_RCR2)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SLEN[4:0] (SPORT Word Length)
00000 - Illegal value
00001 - Illegal value
Serial word length is value in this field plus 1
RRFST (Left/Right Order)
0 - Left stereo channel first
1 - Right stereo channel first
RSFSE (Receive Stereo Frame Sync Enable)
0 - Normal mode
1 - Frame sync becomes L/R clock
RXSE (RxSEC Enable)
0 - Secondary side disabled
1 - Secondary side enabled

Figure 14-28. SPORT Receive Configuration 2 Register

Additional information for the SPORT_RCR1 and SPORTRCR2 receive configuration register bits:

- **Receive enable**. (RSPEN). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the RSPEN bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and receive frame sync pins if so programmed.

  Setting RSPEN enables the SPORT receiver, which can generate a SPORT RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting RSPEN. Setting RSPEN also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting RSPEN.

  Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORT_RCR1 with all of the necessary bits, including RSPEN.

- **Internal receive clock select**. (IRCLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The RCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. (RDTYPE). The two RDTYPE bits specify one of four data formats used for single and multichannel operation.
- **Bit order select**. (RLSBIT). The RLSBIT bit selects the bit order of the data words received over the SPORTs.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word received over the SPORTs) is calculated by adding 1 to the value of the SLEN field. The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field.

The frame sync signal is controlled by the SPORT_TFSDIV and SPORT_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal receive frame sync select**. (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).
- **Receive frame sync required select**. (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a receive frame sync for every data word.

- **Low receive frame sync select**. (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- **Late receive frame sync**. (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.
- **RxSec enable**. (RXSE). This bit enables the receive secondary side of the SPORT (if set).
- **Stereo serial enable**. (RSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (RRFST). If this bit is set, the right channel is received first in stereo serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORT_TCR1, SPORT_TCR2, SPORT_RCR1, and SPORT_RCR2 registers.

SPORT Transmit Data (SPORT_TX) Register

The SPORT_TX register is a write-only register. Reads produce a peripheral bus error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length ≤ 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 14-29. The SPORT_TX register is shown in Figure 14-30 on page 14-57.

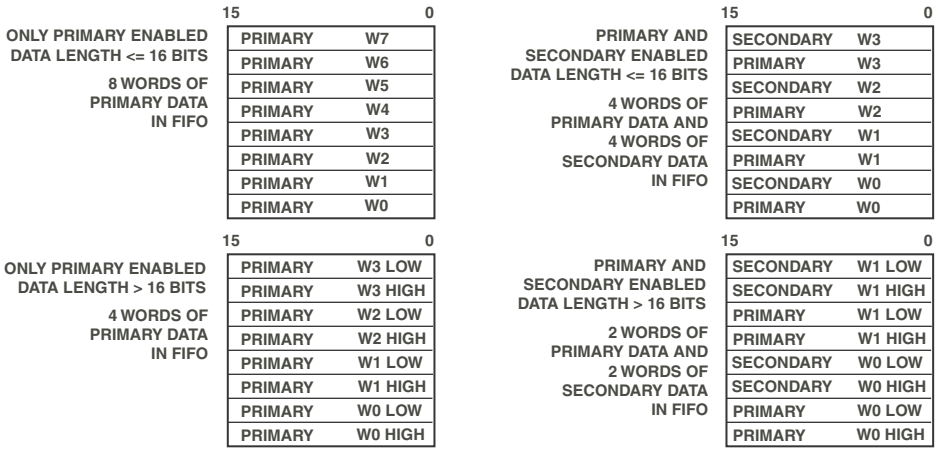


Figure 14-29. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that peripheral bus/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/peripheral bus writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX Hold register based on TXSE and SLEN, and then shifted into the primary

and secondary shift registers. From here, the data is shifted out serially on the DTPRI and DTSEC pins.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled.

The transmit underflow status bit (TUVF) is set in the SPORT_STAT register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORT_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORT_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT_STAT register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORT_STAT register are updated upon writes from the core processor, even when the SPORT is disabled.

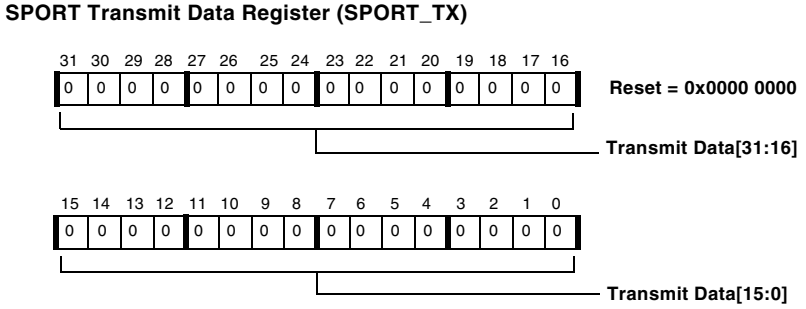


Figure 14-30. SPORT Transmit Data Register

SPORT Receive Data (SPORT_RX) Register

The SPORT_RX register is a read-only register. Writes produce a peripheral bus error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length ≤ 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using peripheral bus/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO are shown in Figure 14-31 on page 14-58. The SPORT_RX register is shown in Figure 14-32 on page 14-60.

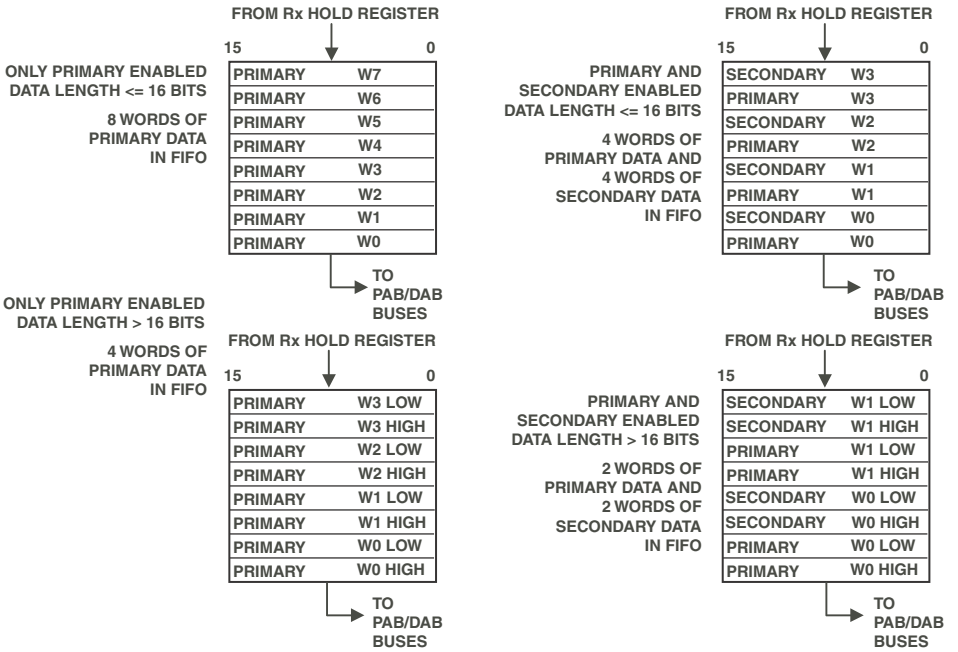


Figure 14-31. SPORT Receive FIFO Data Ordering

When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/peripheral bus reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRPRI pin is loaded into the RX primary shift register, while data from the DRSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted into the RX hold registers for primary and secondary data, respectively. Data from the hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORT_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORT_STAT register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORT_STAT register when a new word is assembled in the RX shift register and the RX hold register has not moved the data to the FIFO. The previously written word in the hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT RX.

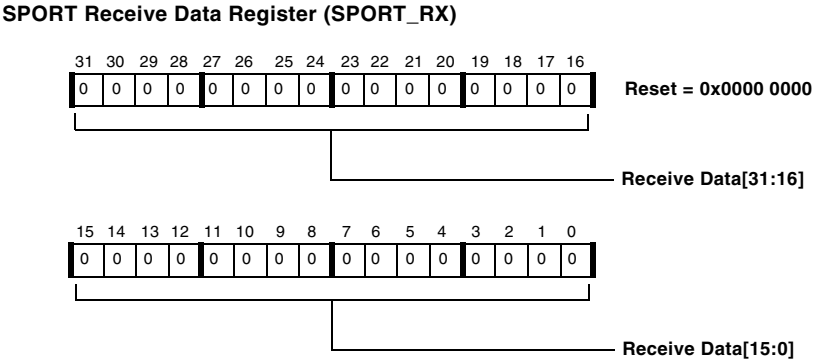


Figure 14-32. SPORT Receive Data Register

SPORT Status (SPORT_STAT) Register

The SPORT_STAT register is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status. This register is shown in Figure 14-33 on page 14-61.

The TXF bit in the SPORT_STAT register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX hold register is empty.

The transmit underflow status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFS may be suppressed whenever SPORT_TX is empty by clearing the DITFS control bit in the SPORT_TCR1 register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TXEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT_STAT register. It is a sticky W1C bit and is also cleared by disabling the SPORT (writing RXEN = 0).

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RXEN = 0. The RUVF bit is updated even when the SPORT is disabled.

SPORT Status Register (SPORT_STAT)

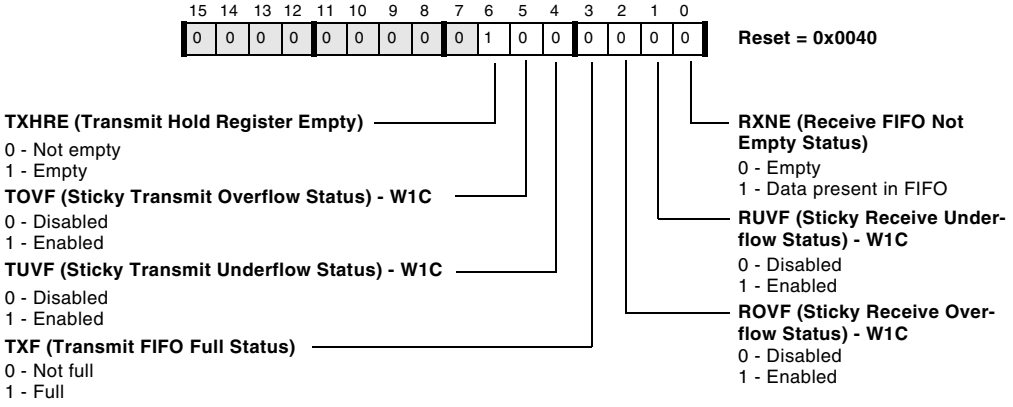

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
Reset = 0x0040
TXHRE (Transmit Hold Register Empty)
0 - Not empty
1 - Empty
TOVF (Sticky Transmit Overflow Status) - W1C
0 - Disabled
1 - Enabled
TUVF (Sticky Transmit Underflow Status) - W1C
0 - Disabled
1 - Enabled
TXF (Transmit FIFO Full Status)
0 - Not full
1 - Full
RXNE (Receive FIFO Not Empty Status)
0 - Empty
1 - Data present in FIFO
RUVF (Sticky Receive Underflow Status) - W1C
0 - Disabled
1 - Enabled
ROVF (Sticky Receive Overflow Status) - W1C
0 - Disabled
1 - Enabled

Figure 14-33. SPORT Status Register

SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORT_TCLKDIV register, shown in Figure 14-34, and the SPORT_RCLKDIV register, shown in Figure 14-35 on page 14-62).

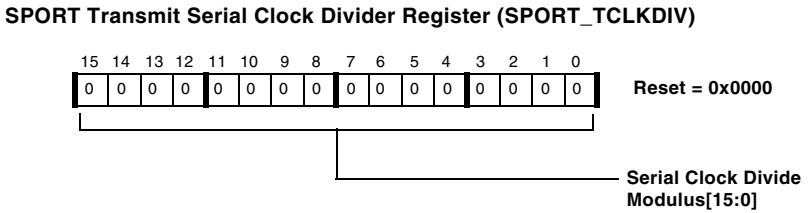

SPORT Transmit Serial Clock Divider Register (SPORT_TCLKDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Serial Clock Divide Modulus[15:0]

Figure 14-34. SPORT Transmit Serial Clock Divider Register

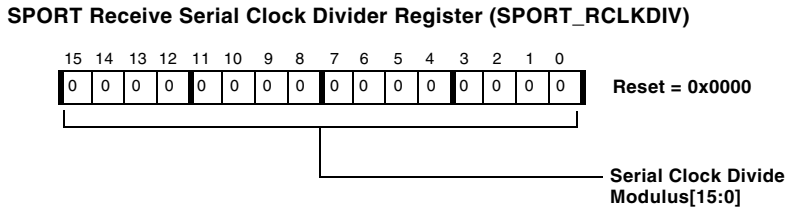

SPORT Receive Serial Clock Divider Register (SPORT_RCLKDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Serial Clock Divide Modulus[15:0]

Figure 14-35. SPORT Receive Serial Clock Divider Register

SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers

The 16-bit SPORT_TFSDIV and SPORT_RFSDIV registers specify how many transmit or receive clock cycles are counted before generating a TFS or RFS pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks. These registers are shown in Figure 14-36 and Figure 14-37 on page 14-63.

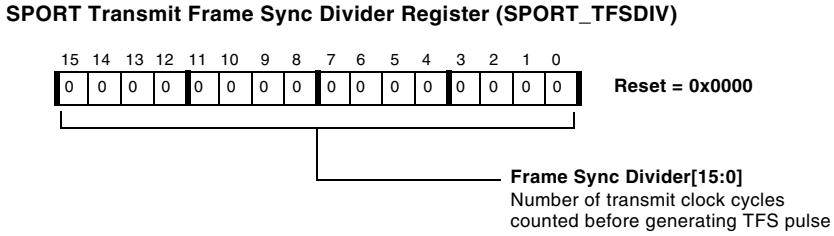

SPORT Transmit Frame Sync Divider Register (SPORT_TFSDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Frame Sync Divider[15:0]
Number of transmit clock cycles
counted before generating TFS pulse

Figure 14-36. SPORT Transmit Frame Sync Divider Register

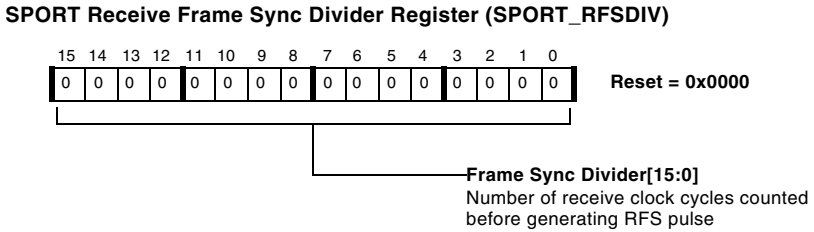

SPORT Receive Frame Sync Divider Register (SPORT_RFSDIV)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
Frame Sync Divider[15:0]
Number of receive clock cycles counted
before generating RFS pulse

Figure 14-37. SPORT Receive Frame Sync Divider Register

SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers

There are two multichannel configuration registers for each SPORT, shown in Figure 14-38 and Figure 14-39 on page 14-64. These registers are used to configure the multichannel operation of the SPORT. The two control registers are shown below.

SPORT Multichannel Configuration Register 1 (SPORT_MCMC1)

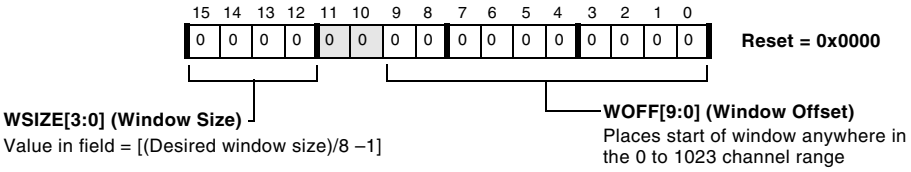

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
WSIZE[3:0] (Window Size)
Value in field = [(Desired window size)/8 –1]
WOFF[9:0] (Window Offset)
Places start of window anywhere in the 0 to 1023 channel range

Figure 14-38. SPORT Multichannel Configuration Register 1

SPORT Multichannel Configuration Register 2 (SPORT_MCMC2)

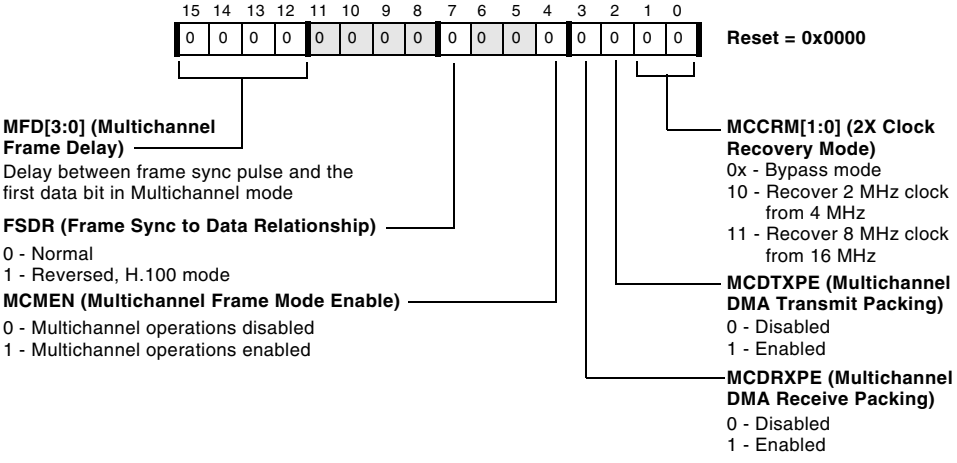

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
MFD[3:0] (Multichannel Frame Delay)
Delay between frame sync pulse and the first data bit in Multichannel mode
FSDR (Frame Sync to Data Relationship)
0 - Normal
1 - Reversed, H.100 mode
MCMEN (Multichannel Frame Mode Enable)
0 - Multichannel operations disabled
1 - Multichannel operations enabled
MCCRM[1:0] (2X Clock Recovery Mode)
0x - Bypass mode
10 - Recover 2 MHz clock from 4 MHz
11 - Recover 8 MHz clock from 16 MHz
MCDTXPE (Multichannel DMA Transmit Packing)
0 - Disabled
1 - Enabled
MCDRXPE (Multichannel DMA Receive Packing)
0 - Disabled
1 - Enabled

Figure 14-39. SPORT Multichannel Configuration Register 2

SPORT Current Channel (SPORT_CHNL) Register

The 10-bit `CHNL` field in the `SPORT_CHNL` register indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The `CHNL[9:0]` field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an

example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between RSCLK and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind. See Figure 14-40 on page 14-65.

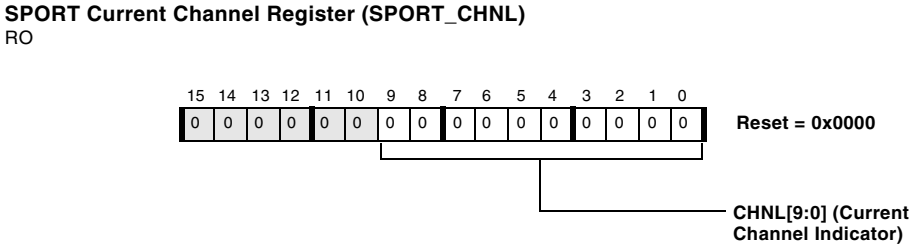

SPORT Current Channel Register (SPORT_CHNL)
RO
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
CHNL[9:0] (Current Channel Indicator)

Figure 14-40. SPORT Current Channel Register

SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers

The SPORT_MRCSn registers (shown in Figure 14-41 on page 14-66) are used to enable and disable individual channels. They specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORT_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the RX buffer. When the secondary receive side is enabled by the RXSE bit, both inputs are processed on enabled channels.

Clearing the bit in the SPORT_MRCSn register causes the SPORT to ignore the data on either channel.

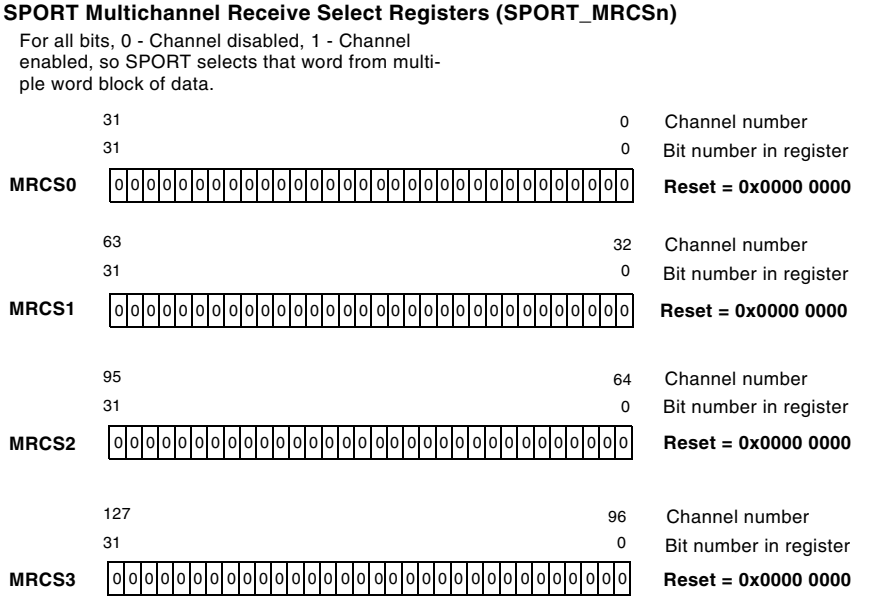


Figure 14-41. SPORT Multichannel Receive Select Registers

SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers

The SPORT_MTCSn registers (shown in Figure 14-42 on page 14-67) are used to enable and disable individual channels. They specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a SPORT_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. When the

secondary transmit side is enabled by the TXSE bit, both sides transmit a word on the enabled channel. Clearing the bit in the SPORT_MTCSn register causes a SPORT controllers' data transmit pins to three-state during the time slot of that channel.

SPORT Multichannel Transmit Select Registers (SPORT_MTCSn)
For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

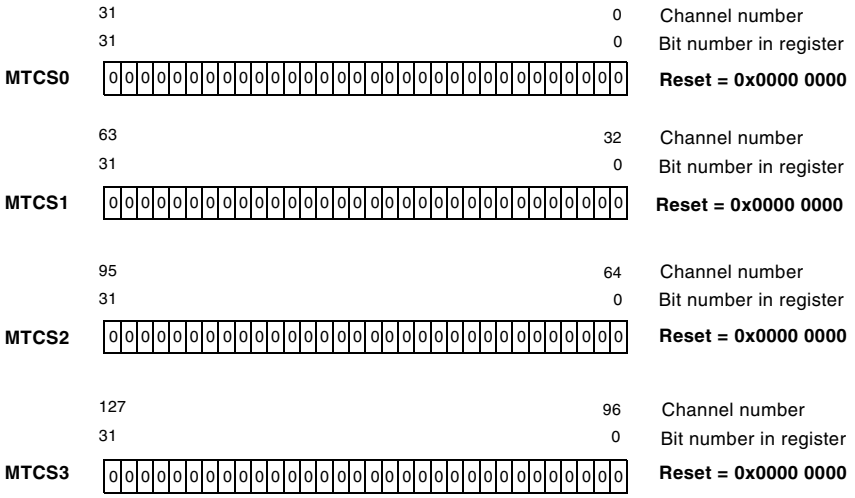


Figure 14-42. SPORT Multichannel Transmit Select Registers

Programming Examples

This section shows an example of typical usage of the SPORT peripheral in conjunction with the DMA controller. See Listing 14-1 on page 14-68 through Listing 14-4 on page 14-73. These listings assume a processor with at least two SPORTs, SPORT0 and SPORT1.

The SPORT is usually employed for high-speed, continuous serial transfers. The example reflects this, in that the SPORT is set-up for auto-buffered, repeated DMA transfers.

Because of the many possible configurations, the example uses generic labels for the content of the SPORT's configuration registers (SPORT_RCRn and SPORT_TCRn) and the DMA configuration. An example value is given in the comments, but for the meaning of the individual bits the user is referred to the detailed explanation in this chapter.

The example configures both the receive and the transmit section. Since they are completely independent, the code uses separate labels.

SPORT Initialization Sequence

The SPORT's receiver and transmitter are configured, but they are not enabled yet.

Listing 14-1. SPORT Initialization

```
Program_SPORT_TRANSMITTER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);
/* Configure Clock speeds */
  R1 = SPORT_TCLK_CONFIG;  /* Divider SCLK/TCLK (value 0 to
65535) */
  W[P0 + (SPORT0_TCLKDIV - SPORT0_TCR1)] = R1;
                        /* TCK divider register */
  /* number of Bitclocks between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_TFSDIV_CONFIG;
  W[P0 + (SPORT0_TFSDIV - SPORT0_TCR1)] = R1;
                       /* TFSDIV register */
/* Transmit configuration */
  /* Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_TRANSMIT_CONF_2;
```

```
  W[P0 + (SPORT0_TCR2 - SPORT0_TCR1)] = R1;
  /* Configuration register 1   (for instance 0x4E12 for inter-
nally generated clk and framesync) */
  R1 = SPORT_TRANSMIT_CONF_1;
  W[P0] = R1;
  ssync;
/* NOTE: SPORT0 TX NOT enabled yet (bit 0 of TCR1 must be zero) */

Program_SPORT_RECEIVER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);
/* Configure Clock speeds */
  R1 = SPORT_RCLK_CONFIG;  /* Divider SCLK/RCLK (value 0 to
65535) */
  W[P0 + (SPORT0_RCLKDIV - SPORT0_RCR1)] = R1;   /* RCK divider
register */
  /* number of Bitclock between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_RFSDIV_CONFIG;
  W[P0 + (SPORT0_RFSDIV - SPORT0_RCR1)] = R1;
                                /* RFSDIV register */
/* Receive configuration */
  /* Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_RECEIVE_CONF_2;
  W[P0 + (SPORT0_RCR2 - SPORT0_RCR1)] = R1;
  /* Configuration register 1   (for instance 0x4410 for external
clk and framesync) */
  R1 = SPORT_RECEIVE_CONF_1;
  W[P0] = R1;
  ssync;  /* NOTE: SPORT0 RX NOT enabled yet (bit 0 of RCR1 must
be zero) */
```

DMA Initialization Sequence

Next the DMA channels for receive (channel3 in this example) and for transmit (channel4 in this example) are set up for auto-buffered, one-dimensional, 32-bit transfers. Again, there are other possibilities, so generic labels have been used, with a particular value shown in the comments.

Note that the DMA channels can be enabled at the end of the configuration since the SPORT is not enabled yet. However, if preferred, the user can enable the DMA later, immediately before enabling the SPORT. The only requirement is that the DMA channel be enabled before the associated peripheral is enabled to start the transfer.

Listing 14-2. DMA Initialization

```
Program_DMA_Controller:
/* Receiver (DMA channel 3) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA3_CONFIG);
  P0.h = hi(DMA3_CONFIG);
  /* Configuration (for instance 0x108A for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_RECEIVE_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* rx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(rx_buf)/4)(z);
  W[P0 + (DMA3_X_COUNT - DMA3_CONFIG)] = R1;
                                        /* X_count register */
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
  W[P0 + (DMA3_X_MODIFY - DMA3_CONFIG)] = R1;
         /* X_modify register */
```

```
            /* start_address register points to memory buffer
                to be filled */
  R1.l = rx_buf;
  R1.h = rx_buf;
  [P0 + (DMA3_START_ADDR - DMA3_CONFIG)] = R1;

  BITSET(R0,0);  /* R0 still contains value of CONFIG register -
set bit 0 */
  W[P0] = R0;    /* enable DMA channel (SPORT not enabled yet) */

/* Transmitter (DMA channel 4) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA4_CONFIG);
  P0.h = hi(DMA4_CONFIG);
  /* Configuration (for instance 0x1088 for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_TRANSMIT_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* tx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(tx_buf)/4)(z);
  W[P0 + (DMA4_X_COUNT - DMA4_CONFIG)] = R1;
              /* X_count register */
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
  W[P0 + (DMA4_X_MODIFY - DMA4_CONFIG)] = R1;
              /* X_modify register */
  /* start_address register points to memory buffer to be
     transmitted from */
  R1.l = tx_buf;
  R1.h = tx_buf;
  [P0 + (DMA4_START_ADDR - DMA4_CONFIG)] = R1;
```

```
  BITSET(R0,0);  /* R0 still contains value of CONFIG register -
                    set bit 0 */
  W[P0] = R0;    /* enable DMA channel (SPORT not enabled yet) */
```

Interrupt Servicing

The receive channel and the transmit channel will each generate an interrupt request if so programmed. The following code fragments show the minimum actions that must be taken. Not shown is the programming of the core and system event controllers.

Listing 14-3. Servicing an Interrupt

```
RECEIVE_ISR:
  [--SP] = RETI;  /* nesting of interrupts */
  /* clear DMA interrupt request */
  P0.h = hi(DMA3_IRQ_STATUS);
  P0.l = lo(DMA3_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear */
  RETI = [SP++];
  rti;
TRANSMIT_ISR:
  [--SP] = RETI;  /* nesting of interrupts */
  /* clear DMA interrupt request */
  P0.h = hi(DMA4_IRQ_STATUS);
  P0.l = lo(DMA4_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear */
  RETI = [SP++];
  rti;
```

Starting a Transfer

After the initialization procedure outlined in the previous sections, the receiver and transmitter are enabled. The core may just wait for interrupts.

Listing 14-4. Starting a Transfer

```
/* Enable Sport0 RX and TX */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /* Enable Receiver (set bit 0) */
  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /* Enable Transmitter (set bit 0) */


/* dummy wait loop (do nothing but waiting for interrupts) */
wait_forever:
 jump wait_forever;
```

Unique Information for the ADSP-BF59x Processor

This section describes Clock Gating Functionality and Modes of Operation that are unique to the ADSP-BF59x processors.

Clock Gating Functionality

A proper interface to many precision A/D converters requires that digital noise be eliminated during conversion quiet zones. These quiet zones can be created by using a gated clock and a guard-banded convert signal waveform. One way to do this is through external logic that properly interfaces the converter to a SPORT on the processor.

The ADSP-BF59x family has integrated “gated clock” hardware that eliminates the need for this external logic and allows the SPORT to interface directly to the converter. This logic constitutes a four-wire interface that is flexible enough to handle many serial A/D converters with different clock rates, converter rates, and converter modes.

Consider an ADC whose digital interface consists of the following four wires:

- `CNV` – convert signal
- `SCK` – data clock
- `DIN` – configuration data input
- `SDO` – conversion result read data

An ADSP-BF59x processor’s SPORT can connect to this converter in the following manner:

- Connect `CNV` to SPORT TFS (transmit frame sync)
- Connect `SCK` to SPORT TSCLK (transmit SPORT clock)
- Connect `DIN` to SPORT DTPRI (primary transmit data)
- Connect `SDO` to SPORT DRPRI (primary receive data)

Key requirements of the converter are that the digital pins remain quiet during critical times of the conversion process. In order to meet this requirement `SCK` must be active only when necessary to read conversion

results and write configuration data. This is what is meant by “gated clock” in this context. In addition, no digital noise should be present in the vicinity of the CNV rising edge. Therefore, the ADSP-BF59x processor can delay assertion of the CNV rising edge by one clock period, to create a guardband between the last SCK edge and the critical CNV rising edge.

Modes of Operation

There are two gated clock modes which are supported. One mode slightly modifies internal SPORT signals to create an external gated clock and CNV signal. The other approach uses timers TMR1 and TMR0 to create arbitrary waveforms for the CNV/TFS signals and gated clocks. In either mode, the clock can be programmed to idle high or low. Both SPORT0 and SPORT1 can support gated clock operation, and the chosen modes of operation for each SPORT are completely independent.

Gated Clock Mode 0 – SPORT Gated Clocks Without Using TIMERs

In this mode, the internal TFS is AND’ed (taking polarities into account) with TSCLK to create a gated clock. The external TFS pin is just the internal TFS extended by one clock cycle to create the CNV signal. The internal TFS signal is looped back to RFS (receive frame sync) to allow correct receive SPORT operation. Configuration data is transmitted on the DTPRI pin at the same time conversion results are received on the DRPRI pin.

Gated Clock Mode 1 – SPORT Gated Clocks Using TIMERs

In this mode, the SPORT clock is used as the PWM_CLK source for both TMR0 and TMR1. TMR0 is programmed to create a valid TFS waveform and TMR1 is programmed to create a valid CNV waveform. The TMR0 output is AND’ed (taking polarities into account) with TSCLK to created a gated clock. The timers are simultaneously enabled via the TIMER_ENABLE register to ensure that the edges are perfectly synchronized. As in Gated Clock Mode 0, the TFS signal is looped back to RFS to allow correct

receive SPORT operation. Again, configuration data is transmitted on the DTPRI pin at the same time conversion results are received on the DRPRI pin.

Programming Model

To enable the gated clock modes, refer to the bit definitions located in the SPORT Clock Gating register (shown in Figure 14-43). No special pin-multiplexing configuration is required to enable the gated clock modes. The gated clock appears on the TSCLK pin of the respective SPORT.

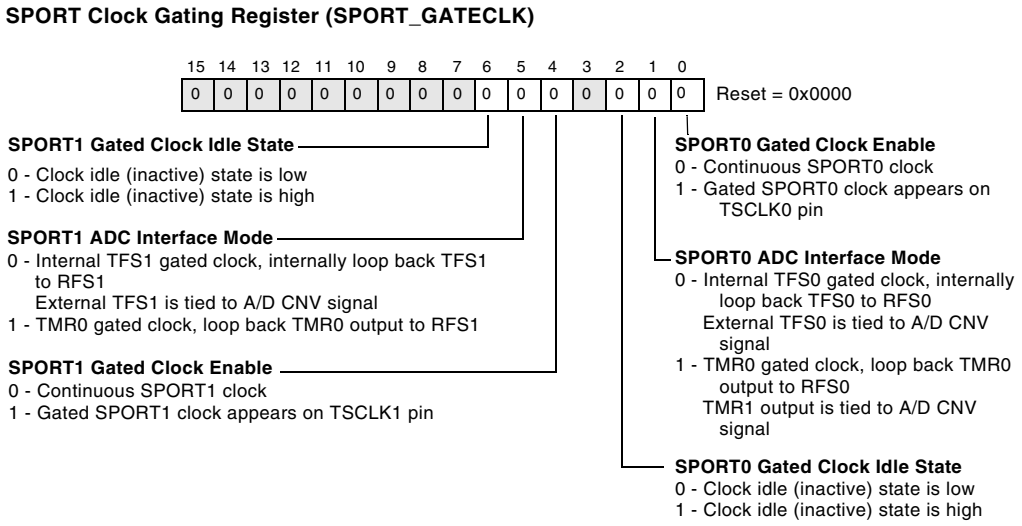

SPORT Clock Gating Register (SPORT_GATECLK)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
SPORT1 Gated Clock Idle State
0 - Clock idle (inactive) state is low
1 - Clock idle (inactive) state is high
SPORT1 ADC Interface Mode
0 - Internal TFS1 gated clock, internally loop back TFS1 to RFS1
External TFS1 is tied to A/D CNV signal
1 - TMR0 gated clock, loop back TMR0 output to RFS1
SPORT1 Gated Clock Enable
0 - Continuous SPORT1 clock
1 - Gated SPORT1 clock appears on TSCLK1 pin
SPORT0 Gated Clock Enable
0 - Continuous SPORT0 clock
1 - Gated SPORT0 clock appears on TSCLK0 pin
SPORT0 ADC Interface Mode
0 - Internal TFS0 gated clock, internally loop back TFS0 to RFS0
External TFS0 is tied to A/D CNV signal
1 - TMR0 gated clock, loop back TMR0 output to RFS0
TMR1 output is tied to A/D CNV signal
SPORT0 Gated Clock Idle State
0 - Clock idle (inactive) state is low
1 - Clock idle (inactive) state is high

Figure 14-43. SPORT Clock Gating Register

Figure 14-44 shows an example of the ADSP-BF59x to AD71090 interface with a 25 MHz data clock and conversion rate of approximately 390 kSPS. In this example, the SPORT is connected in ADC Interface Mode 0.

In Figure 14-44, note the following:

- SDO read data is driven by the CNV falling edge (MSB) or the SCK falling edges.
- DIN configuration data is driven on TSCLK falling edges and sampled on SCK rising edges.
- Internal RFS delayed internally by one-half clock period for proper SPORT timing.
- CNV rising edge is delayed by two clock cycles relative to the internal TFS.
- SDO cannot be three-stated (needs a pull-up resistor).

Also in Figure 14-44, note the following for SPORT configuration (for 25 MHz TSCLK/RSCLK):

- TFSDIV = 64 cycles
- SLEN = 16 bits
- TCKFE = 1 (falling edge drive)
- RCKFE = 0 (sample SDO on falling edge)
- LATFS = 1 (late frame sync)
- LTFS = 1 (active low TFS)

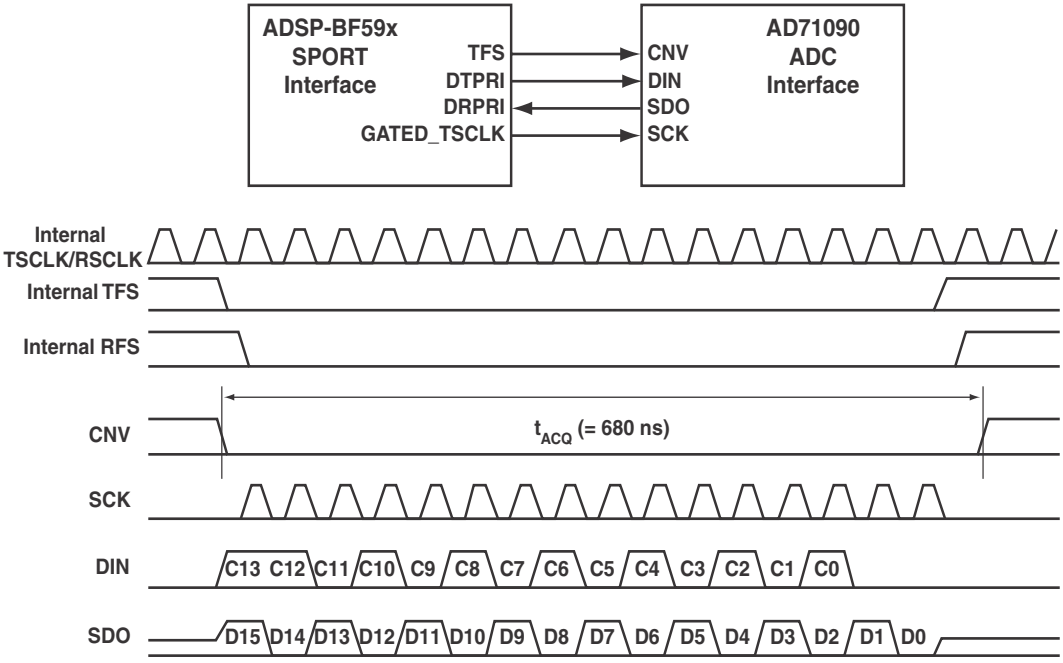


Figure 14-44. ADSP-BF59x to AD71090 Interface Timing (@ 25 MHz)

15 PARALLEL PERIPHERAL INTERFACE

This chapter describes the parallel peripheral interface (PPI). Following an overview and a list of key features are a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Specific Information for the ADSP-BF59x

For details regarding the number of PPIs for the ADSP-BF59x product, please refer to the *ADSP-BF592 Blackfin Processor Data Sheet*.

For PPI DMA channel assignments, refer to Table 5-9 on page 5-103 in Chapter 5, “Direct Memory Access”.

For PPI interrupt vector assignments, refer to Table 4-3 on page 4-17 in Chapter 4, “System Interrupts”.

To determine how each of the PPIs is multiplexed with other functional pins, refer to Table 7-1 on page 7-3 through Table 7-2 on page 7-4 in Chapter 7, “General-Purpose Ports”.

For a list of MMR addresses for each PPI, refer to Chapter A, “System MMR Assignments”.

PPI behavior for the ADSP-BF59x that differs from the general information in this chapter can be found in the section “Unique Information for the ADSP-BF59x Processor” on page 15-37

Overview

The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin and three multiplexed frame sync pins. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

Features

The PPI includes these features:

- Half duplex, bidirectional parallel port
- Supports up to 16 bits of data
- Programmable clock and frame sync polarities
- ITU-R 656 support
- Interrupt generation on overflow and underrun

Typical peripheral devices that can be interfaced to the PPI port:

- A/D converters
- D/A converters
- LCD panels
- CMOS sensors
- Video encoders
- Video decoders

Interface Overview

Figure 15-1 shows a block diagram of the PPI.

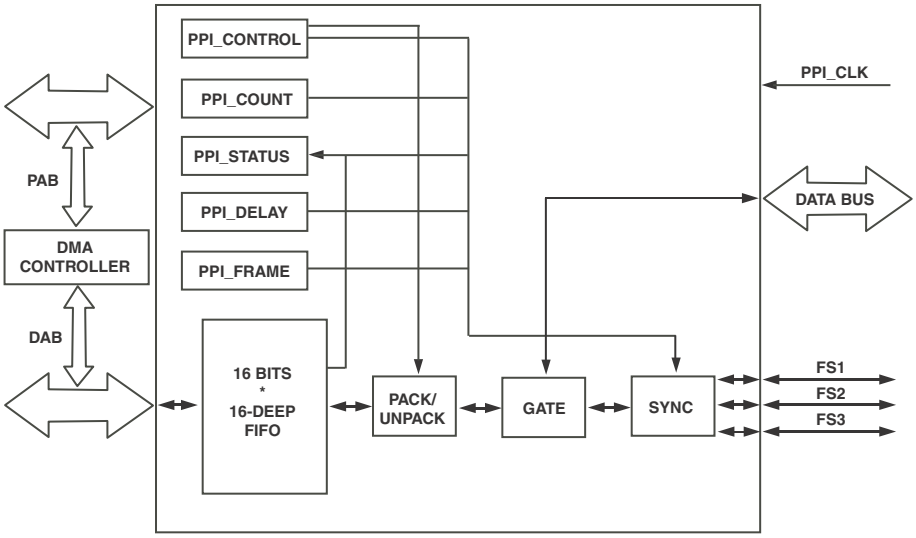


Figure 15-1. PPI Block Diagram

The PPI_CLK pin accepts an external clock input. It cannot source a clock internally.

When the PPI_CLK is not free-running, there may be additional latency cycles before data gets received or transmitted. In RX and TX modes, there may be at least 2 cycles latency before valid data is received or transmitted.

The PPI_CLK not only supplies the PPI module itself, but it also can clock one or more GP Timers to work synchronously with the PPI. Depending on PPI operation mode, the PPI_CLK can either equal or invert the TMRCLK input. For more information, see Chapter 8, “General-Purpose Timers”.

Description of Operation

Table 15-1 shows all the possible modes of operation for the PPI.

Table 15-1. PPI Possible Operating Modes

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_ SEL
RX mode, 0 frame syncs, external trigger	0	0	11	11	0 or 1	0 or 1	0
RX mode, 0 frame syncs, internal trigger	0	0	11	11	0 or 1	0 or 1	1
RX mode, 1 external frame sync	1	0	00	11	0 or 1	0 or 1	0
RX mode, 2 or 3 external frame syncs	3	0	10	11	0 or 1	0 or 1	0
RX mode, 2 or 3 internal frame syncs	3	0	01	11	0 or 1	0 or 1	0
RX mode, ITU-R 656, active field only	embedded	0	00	00	0 or 1	0	0 or 1
RX mode, ITU-R 656, vertical blanking only	embedded	0	00	10	0 or 1	0	0
RX mode, ITU-R 656, entire field	embedded	0	00	01	0 or 1	0	0
TX mode, 0 frame syncs	0	1	00	00	0 or 1	0 or 1	0
TX mode, 1 internal or external frame sync	1	1	00	11	0 or 1	0 or 1	0
TX mode, 2 external frame syncs	2	1	01	11	0 or 1	0 or 1	0

Table 15-1. PPI Possible Operating Modes (Continued)

PPI Mode	# of Syncs	PORT_DIR	PORT_CFG	XFR_TYPE	POLC	POLS	FLD_ SEL
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS1 assertion	3	1	01	11	0 or 1	0 or 1	0
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS2 assertion	3	1	11	11	0 or 1	0 or 1	0

Functional Description

The following sections describe the function of the PPI.

ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 15-2, and Figure 15-3 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the horizontal (H), vertical (V), and field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The start of active video (SAV) and end of active video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a

0-to-1 transition of H. An entire field of video is comprised of active video + horizontal blanking (the space between an EAV and SAV code) and vertical blanking (the space where V = 1). A field of video commences on a transition of the F bit. The “odd field” is denoted by a value of F = 0, whereas F = 1 denotes an even field. Progressive video makes no distinction between field 1 and field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

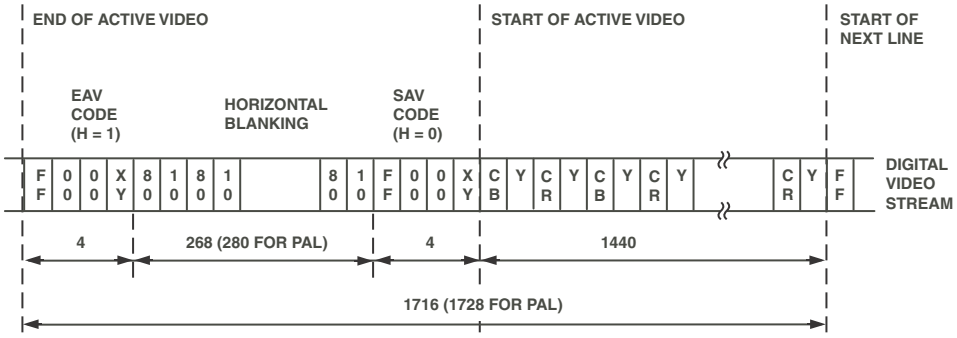
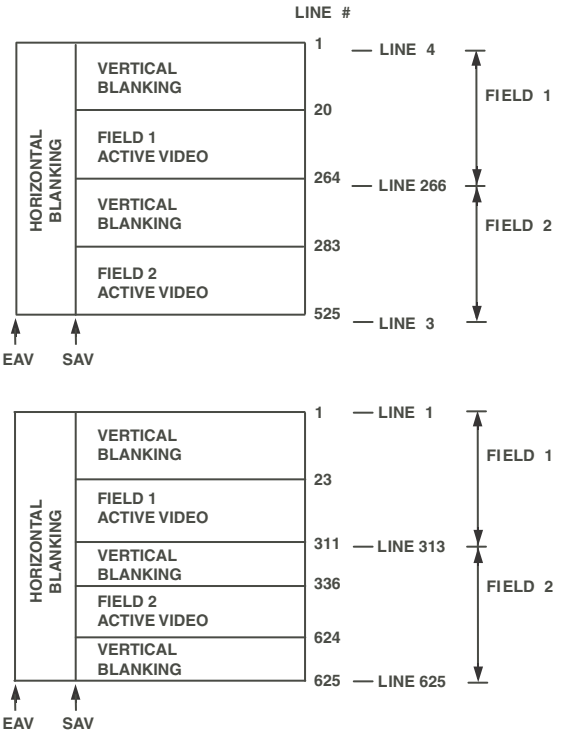

END OF ACTIVE VIDEO
START OF ACTIVE VIDEO
START OF NEXT LINE
EAV CODE (H = 1)
HORIZONTAL BLANKING
SAV CODE (H = 0)
FF 00 00 XY 80 10 80 10
80 10 FF 00 00 XY CB Y CR Y CB Y CR Y
CR Y FF
DIGITAL VIDEO STREAM
4
268 (280 FOR PAL)
4
1440
1716 (1728 FOR PAL)

Figure 15-2. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0

LINE NUMBER	F	V	H (EAV)	H (SAV)
1-22, 311-312	0	1	1	0
23-310	0	0	1	0
313-335, 624-625	1	1	1	0
336-623	1	0	1	0

Figure 15-3. Typical Video Frame Partitioning for NTSC/PAL Systems for ITU-R BT.656-4

The SAV and EAV codes are shown in more detail in Table 15-2. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY status word, which, aside from the F (field), V (vertical blanking) and H (horizontal blanking) bits, contains four protection bits for single-bit error detection and correction. Note F and V are only allowed to change as part of EAV sequences (that is, transition from H = 0 to H = 1). The bit definitions are as follows:

- F = 0 for field 1
- F = 1 for field 2

- V = 1 during vertical blanking
- V = 0 when not in vertical blanking
- H = 0 at SAV
- H = 1 at EAV
- P3 = V XOR H
- P2 = F XOR H
- P1 = F XOR V
- P0 = F XOR V XOR H

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be "656-compliant," where EAV and SAV values define the range of the image for each line, and the V and F codes can be used to delimit fields and frames.

Table 15-2. Control Byte Sequences for 8-bit and 10-bit ITU-R 656 Video

	8-bit Data								10-bit Data	
	D9 (MSB)	D8	D7	D6	D5	D4	D3	D2	D1	D0
Preamble	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
Control Byte	1	F	V	H	P3	P2	P1	P0	0	0

ITU-R 656 Input Modes

Figure 15-4 shows a general illustration of data movement in the ITU-R 656 input modes. In the figure, the clock `CLK` is either provided by the video source or supplied externally by the system.

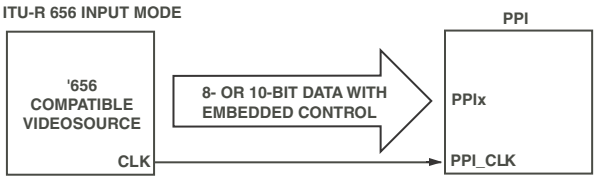


Figure 15-4. ITU-R 656 Input Modes

There are three submodes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 15-5 shows these three submodes.

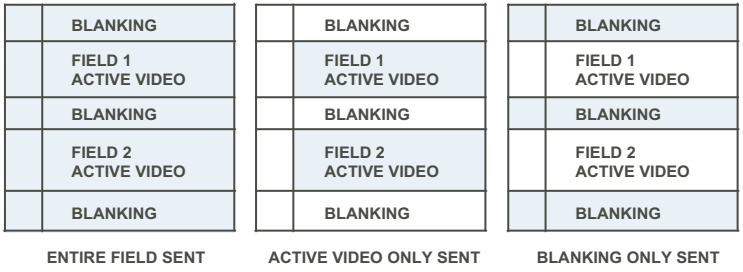


Figure 15-5. ITU-R 656 Input Submodes

Entire Field

In this mode, the entire incoming bitstream is read in through the PPI. This includes active video as well as control byte sequences and ancillary data that may be embedded in horizontal and vertical blanking intervals. Data transfer starts immediately after synchronization to field 1 occurs,

but does not include the first EAV code that contains the F = 0 assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a "loopback" feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.

Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when V = 1. In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the PPI_FRAME MMR.

Vertical Blanking Interval (VBI) only

In this mode, data transfer is only active while V = 1 is in the control byte sequence. This indicates that the video source is in the midst of the vertical blanking interval (VBI), which is sometimes used for ancillary data transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI. Control byte sequence information is always logged. The user specifies the number of total lines (active plus vertical blanking) per frame in the PPI_FRAME MMR.

Note the VBI is split into two regions within each field. From the PPI's standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of field 1, which doesn't necessarily correspond to the start of vertical blanking. For instance, in 525/60 systems, the start of field 1 (F = 0) corresponds to line 4 of the VBI.

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. However, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of field 1. Consequently, up to two fields might be ignored (for example, if field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signaling is embedded in the datastream in ITU-R 656 modes, the PPI_COUNT register is not necessary. However, the PPI_FRAME register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that

occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in PPI_FRAME. If there is a mismatch, the FT_ERR bit in the PPI_STATUS register is asserted. For instance, if an SAV transition is missed, the current field will only have NUM_ROWS - 1 rows, but resynchronization will reoccur at the start of the next frame.

Upon completing reception of an entire field, the field status bit is toggled in the PPI_STATUS register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose PPI Modes

The general-purpose PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 15-3 summarizes these modes. If a particular mode shows a given PPI_FSx frame sync not being used, this implies that the pin is available for its alternate, multiplexed functions.

Table 15-3. General-Purpose PPI Modes

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 0 frame syncs, external trigger	Input	Not used	Not used	Input
RX mode, 0 frame syncs, internal trigger	Not used	Not used	Not used	Input
RX mode, 1 external frame sync	Input	Not used	Not used	Input
RX mode, 2 or 3 external frame syncs	Input	Input	Input (if used)	Input
RX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Input
TX mode, 0 frame syncs	Not used	Not used	Not used	Output
TX mode, 1 external frame sync	Input	Not used	Not used	Output

Table 15-3. General-Purpose PPI Modes (Continued)

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
TX mode, 2 external frame syncs	Input	Input	Not used	Output
TX mode, 1 internal frame sync	Output	Not used	Not used	Output
TX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Output

Figure 15-6 illustrates the general flow of the general purpose PPI modes. The top of the diagram shows an example of RX mode with one external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT. Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter reinitializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 15-6 shows an example of TX mode, one internal frame sync. After PPI_FS1 is asserted, there is a latency of one PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into PPI_DELAY. Next, the DMA controller transfers out the number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

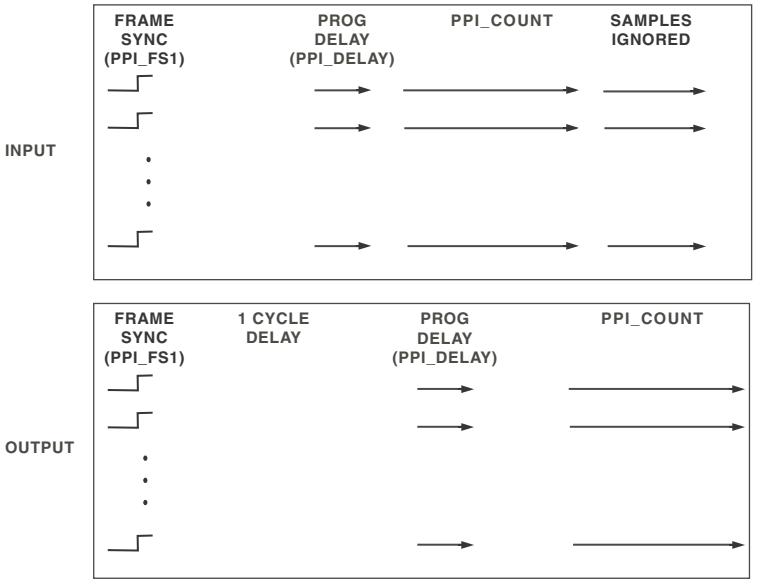


Figure 15-6. General Flow for GP Modes (Assumes Positive Assertion of PPI_FS1)

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 15-1 on page 15-4 for information on how to configure the PPI for each mode.

No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.

- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K byte memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least two PPI_CLK cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The frame syncs are level-sensitive signals. The 1-sync mode is intended for analog-to-digital converter (ADC) applications. The top part of Figure 15-7 shows a typical illustration of the system setup for this mode.

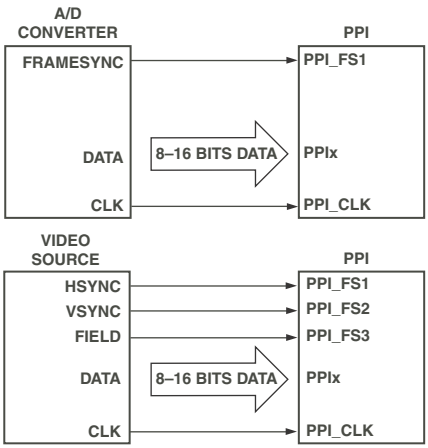


Figure 15-7. RX Mode, External Frame Syncs

The 3-sync mode shown at the bottom of Figure 15-7 supports video applications that use hardware signaling (HSYNC, VSYNC, FIELD) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is PPI_FS1 = HSYNC, PPI_FS2 = VSYNC, PPI_FS3 = FIELD. Please refer to "Frame Synchronization in GP Modes" on page 15-19 for more information about frame syncs in this mode.

A 2-sync mode is supported by not enabling the PPI_FS3 pin. See the *Product Specific Implementation* section for information on how this is achieved on this processor.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read from the video source by asserting PPI_FS1 and PPI_FS2, and then reading data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 15-8 shows a sample application for this mode.

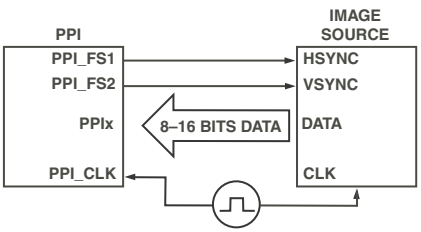


Figure 15-8. RX Mode, Internal Frame Syncs

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 15-1 on page 15-4 for information on how to configure the PPI for each mode.

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers will take place immediately, synchronized to PPI_CLK. See Figure 15-9 for an illustration of this mode.

In this mode, there is a delay of up to 16 SCLK cycles (for > 8-bit data) or 32 SCLK cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

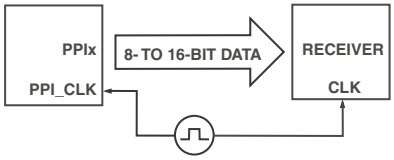


Figure 15-9. TX Mode, 0 Frame Syncs

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in Figure 15-10 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of 1.5 PPI_CLK cycles, plus the value programmed in PPI_DELAY, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.

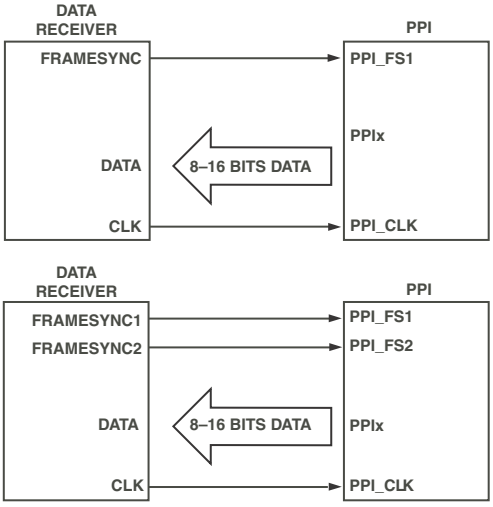


Figure 15-10. TX Mode, 1 or 2 External Frame Syncs

1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to digital-to-analog converters (DACs) with a single frame sync. The top part of Figure 15-11 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 15-11. A 2-sync mode is implicitly supported by leaving `PPI_FS3` unconnected in this case.

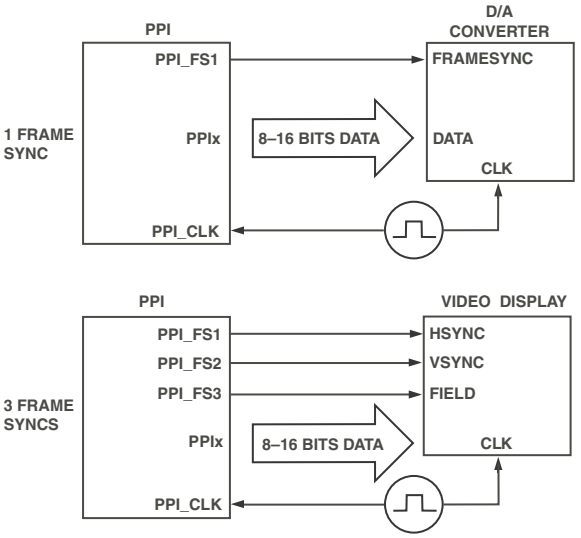


Figure 15-11. PPI GP Output

Frame Synchronization in GP Modes

Frame synchronization in general purpose modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes With Internal Frame Syncs

In modes with internal frame syncs, PPI_FS1 and PPI_FS2 link directly to the pulsewidth modulation (PWM) circuits of general purpose timers. See the Chapter 8, “General-Purpose Timers” for information on how this is achieved on this processor. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing TIMERx registers. This capability accommodates a wide range of timing needs. Note these PWM circuits are clocked by PPI_CLK, not by SCLK (as during conventional timer PWM operation). If PPI_FS2 is not used in the configured PPI mode, its corresponding timer operates as it normally would, unrestricted in functionality. The state of PPI_FS3 depends completely on the

state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:

1. Configure and enable DMA for the PPI. See "DMA Operation" on page 15-22.
2. Configure the width and period for each frame sync signal via the appropriate TIMER_WIDTH and TIMER_PERIOD registers.
3. Set up the appropriate TIMER_CONFIG register(s) for PWM_OUT mode. This includes setting CLK_SEL to 1 and TIN_SEL to 1 for each timer involved.
4. Write to PPI_CONTROL to configure and enable the PPI.
5. Write to TIMER_ENABLE to enable the appropriate timer(s).

It is important to guarantee proper frame sync polarity between the PPI and timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in the appropriate TIMER_CONFIG register(s). Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in the appropriate TIMER_CONFIG register(s).

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).
2. Disable the appropriate timer(s) (using TIMER_DISABLE).

Modes With External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such modes the timers associated with the PPI_FS1 and PPI_FS2 pins can still be used for a purpose not involving the actual pin. However, timer access to a TMRx pin is disabled when the

PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, the associated timer is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR1 pin becomes available for timer use as well). For more information on configuring and using the timers, please refer to the *General-Purpose Timers* chapter.

In RX mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the timebase (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMER_CONFIG. Then the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see the *General-Purpose Timers* chapter.

Programming Model

The following sections describe the PPI programming model.

DMA Operation

The PPI must be used with the processor's DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, please refer to the *Direct Memory Access* chapter.

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of (PPI_CLK) × (16 bits/transfer). In modes where data lengths are greater than eight bits, only one element can be clocked in per PPI_CLK cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and PACK_EN = 1 (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI's DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA error occurs. In fact, the specification of the DMA_XCOUNT and DMA_YCOUNT MMRs allows for flexible data interrupt points. For example, assume the DMA registers XMODIFY = YMODIFY = 1. Then, if a data frame contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 1 (the DI_SEL bit is located in DMA_CONFIG) interrupts on every row transferred, for the entire frame.
- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 0 interrupts only on the completion of the frame (when 240 rows of 320 bytes have been transferred).

- Setting XCOUNT = 38,400 (320 x 120), YCOUNT = 2, and DI_SEL = 1 causes an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

The general procedure for setting up DMA operation with the PPI follows.

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a 1 to bit 0 in PPI_CONTROL.
5. If internally generated frame syncs are used, write to the TIMER_ENABLE register to enable the timers linked to the PPI frame syncs.

Figure 15-12 shows a flow diagram detailing the steps on how to configure the PPI for the various modes of operation.

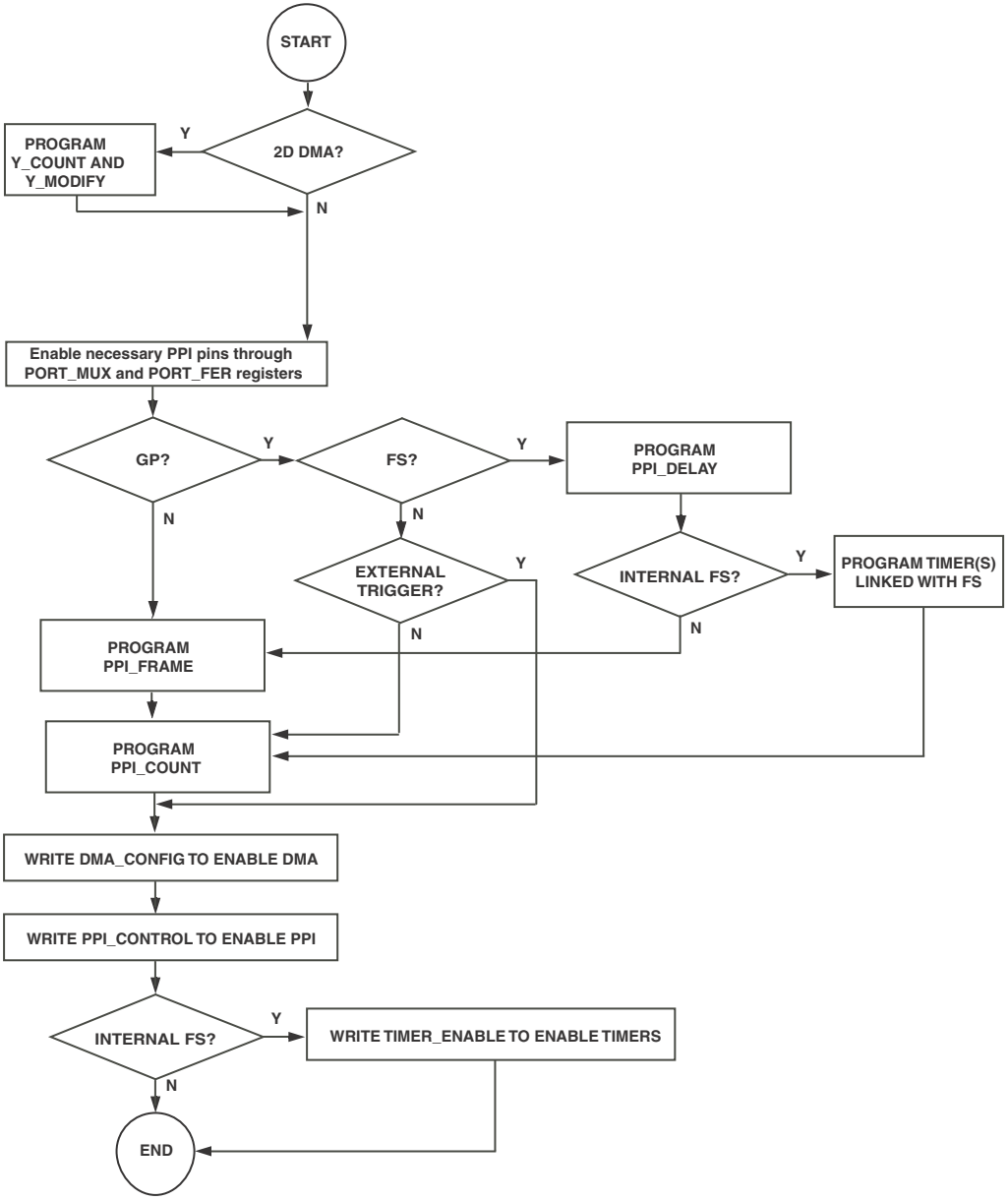


Figure 15-12. PPI Flow Diagram

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI control register (PPI_CONTROL), the PPI status register (PPI_STATUS), the delay count register (PPI_DELAY), the transfer count register (PPI_COUNT), and the lines per frame register (PPI_FRAME).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

PPI Control Register (PPI_CONTROL)

The PPI_CONTROL register configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 15-13 for a bit diagram of this MMR.

The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FS1/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any pins unused by the PPI as a result of the DLEN setting are free for use in their other functions.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

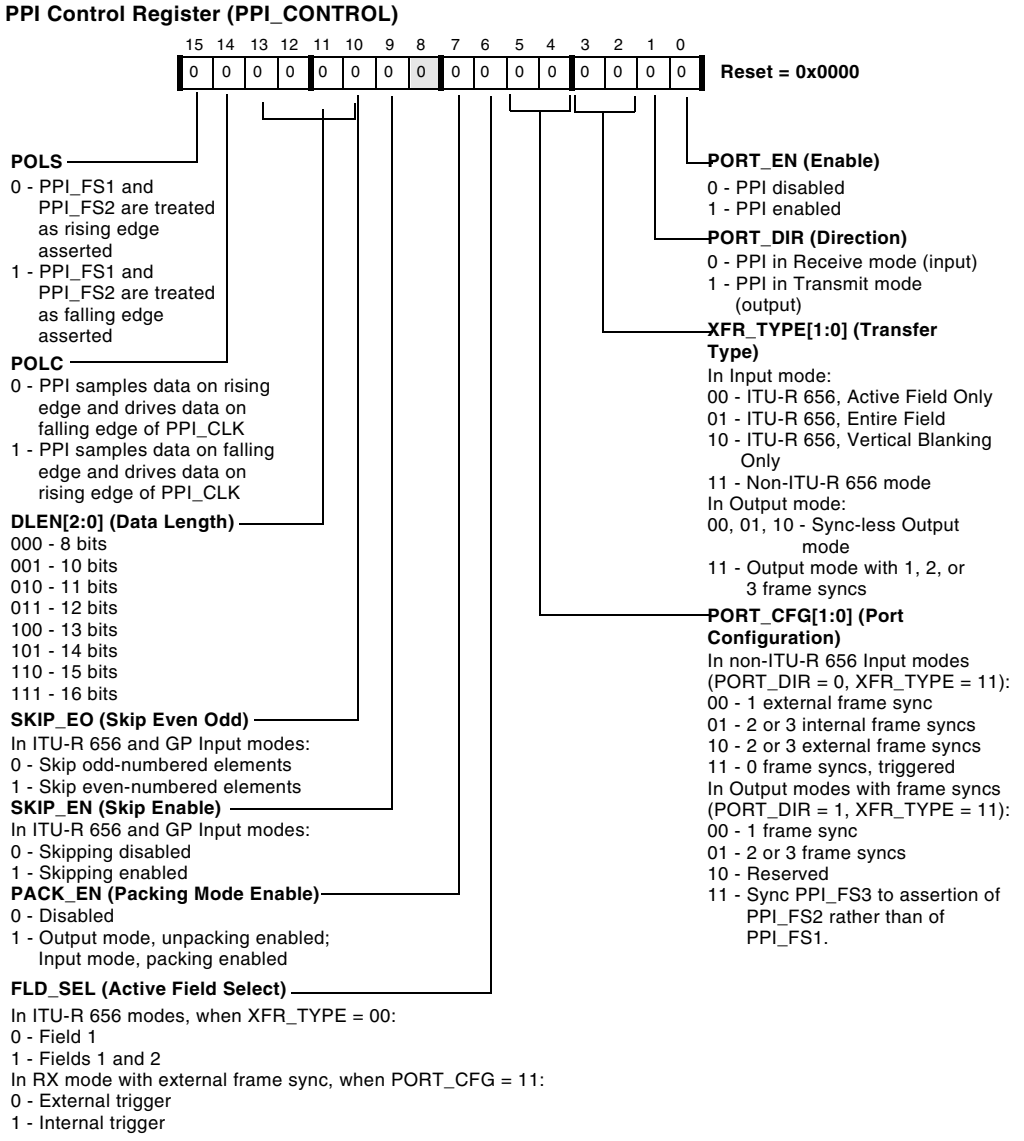

PPI Control Register (PPI_CONTROL)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
POLS
0 - PPI_FS1 and PPI_FS2 are treated as rising edge asserted
1 - PPI_FS1 and PPI_FS2 are treated as falling edge asserted
POLC
0 - PPI samples data on rising edge and drives data on falling edge of PPI_CLK
1 - PPI samples data on falling edge and drives data on rising edge of PPI_CLK
DLEN[2:0] (Data Length)
000 - 8 bits
001 - 10 bits
010 - 11 bits
011 - 12 bits
100 - 13 bits
101 - 14 bits
110 - 15 bits
111 - 16 bits
SKIP_EO (Skip Even Odd)
In ITU-R 656 and GP Input modes:
0 - Skip odd-numbered elements
1 - Skip even-numbered elements
SKIP_EN (Skip Enable)
In ITU-R 656 and GP Input modes:
0 - Skipping disabled
1 - Skipping enabled
PACK_EN (Packing Mode Enable)
0 - Disabled
1 - Output mode, unpacking enabled; Input mode, packing enabled
FLD_SEL (Active Field Select)
In ITU-R 656 modes, when XFR_TYPE = 00:
0 - Field 1
1 - Fields 1 and 2
In RX mode with external frame sync, when PORT_CFG = 11:
0 - External trigger
1 - Internal trigger
PORT_EN (Enable)
0 - PPI disabled
1 - PPI enabled
PORT_DIR (Direction)
0 - PPI in Receive mode (input)
1 - PPI in Transmit mode (output)
XFR_TYPE[1:0] (Transfer Type)
In Input mode:
00 - ITU-R 656, Active Field Only
01 - ITU-R 656, Entire Field
10 - ITU-R 656, Vertical Blanking Only
11 - Non-ITU-R 656 mode
In Output mode:
00, 01, 10 - Sync-less Output mode
11 - Output mode with 1, 2, or 3 frame syncs
PORT_CFG[1:0] (Port Configuration)
In non-ITU-R 656 Input modes (PORT_DIR = 0, XFR_TYPE = 11):
00 - 1 external frame sync
01 - 2 or 3 internal frame syncs
10 - 2 or 3 external frame syncs
11 - 0 frame syncs, triggered
In Output modes with frame syncs (PORT_DIR = 1, XFR_TYPE = 11):
00 - 1 frame sync
01 - 2 or 3 frame syncs
10 - Reserved
11 - Sync PPI_FS3 to assertion of PPI_FS2 rather than of PPI_FS1.

Figure 15-13. PPI Control Register

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input datastream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor's) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.

The PACK_EN bit only has meaning when the PPI port width (selected by DLEN[2:0]) is eight bits. Every PPI_CLK-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of ten bits still results in a 16-bit input word for every PPI_CLK; the upper 6 bits are 0s. Likewise, a port width of eight bits also results in a 16-bit input word, with the upper eight bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the PACK_EN bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:

```
0xCE, 0xFA, 0xFE, 0xCA....
```

- With PACK_EN set:

  This is read into the PPI, configured for an 8-bit port width:

  ```
  0xCE, 0xFA, 0xFE, 0xCA...
  ```

  This is transferred onto the DMA bus:

  ```
  0xFACE, 0xCAFE,...
  ```

- With PACK_EN cleared:

  This is read into the PPI:

  ```
  0xCE, 0xFA, 0xFE, 0xCA,...
  ```

  This is transferred onto the DMA bus:

  ```
  0x00CE, 0x00FA, 0x00FE, 0x00CA,...
  ```

For TX modes, setting PACK_EN enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:

```
0xFACE CAFE....
```

(0xFA and 0xCA are the two most significant bits (MSBs) of their respective 16-bit words)

- With PACK_EN set:

  This is DMAed to the PPI:

  ```
  0xFACE, 0xCAFE,...
  ```

  This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first):

  ```
  0xCE, 0xFA, 0xFE, 0xCA,...
  ```

- With PACK_EN cleared:

  This is DMAed to the PPI:

  ```
  0xFACE, 0xCAFE,...
  ```

  This is transferred out through the PPI, configured for an 8-bit port width:

  ```
  0xCE, 0xFE,...
  ```

The FLD_SEL bit is used primarily in the active field only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only field 1 of each video frame, or both fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 15-1 on page 15-4 for the possible operating modes for the PPI.

The XFR_TYPE[1:0] field configures the PPI for various modes of operation. Refer to Table 15-1 on page 15-4 to see how XFR_TYPE[1:0] interacts with other bits in PPI_CONTROL to determine the PPI operating mode.

The PORT_EN bit, when set, enables the PPI for operation.

When configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section "Frame Synchronization in GP Modes" on page 15-19 for more information.

PPI Status Register (PPI_STATUS)

The PPI_STATUS register, shown in Figure 15-14, contains bits that provide information about the current operating state of the PPI.

The ERR_DET bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in ITU-R 656 modes. If ERR_DET = 1, an error was detected in the preamble. If ERR_DET = 0, no error was detected in the preamble.

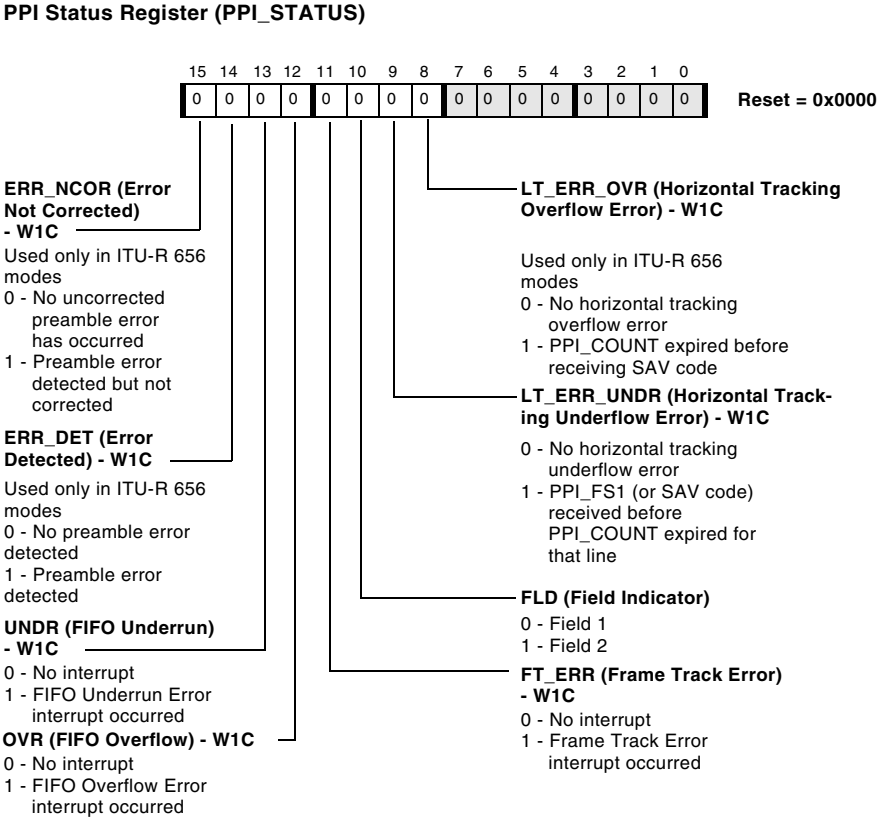

PPI Status Register (PPI_STATUS)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
ERR_NCOR (Error Not Corrected) - W1C
Used only in ITU-R 656 modes
0 - No uncorrected preamble error has occurred
1 - Preamble error detected but not corrected
ERR_DET (Error Detected) - W1C
Used only in ITU-R 656 modes
0 - No preamble error detected
1 - Preamble error detected
UNDR (FIFO Underrun) - W1C
0 - No interrupt
1 - FIFO Underrun Error interrupt occurred
OVR (FIFO Overflow) - W1C
0 - No interrupt
1 - FIFO Overflow Error interrupt occurred
LT_ERR_OVR (Horizontal Tracking Overflow Error) - W1C
Used only in ITU-R 656 modes
0 - No horizontal tracking overflow error
1 - PPI_COUNT expired before receiving SAV code
LT_ERR_UNDR (Horizontal Tracking Underflow Error) - W1C
0 - No horizontal tracking underflow error
1 - PPI_FS1 (or SAV code) received before PPI_COUNT expired for that line
FLD (Field Indicator)
0 - Field 1
1 - Field 2
FT_ERR (Frame Track Error) - W1C
0 - No interrupt
1 - Frame Track Error interrupt occurred

Figure 15-14. PPI Status Register

The ERR_NCOR bit is sticky and is relevant only in ITU-R 656 modes. If ERR_NCOR = 0 and ERR_DET = 1, all preamble errors that have occurred have been corrected. If ERR_NCOR = 1, an error in the preamble was detected but not corrected. This situation generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FT_ERR bit is sticky and indicates, when set, that a frame track error has occurred. In this condition, the programmed number of lines per frame in PPI_FRAME does not match up with the “frame start detect” condition (see the information note on page 15-34). A frame track error

generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FLD bit is set or cleared at the same time as the change in state of F (in ITU-R 656 modes) or PPI_FS3 (in other RX modes). It is valid for input modes only. The state of FLD reflects the current state of the F or PPI_FS3 signals. In other words, the FLD bit always reflects the current video field being processed by the PPI.

The OVR bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO overflow error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The PPI FIFO is 16 bits wide and has 16 entries.

The UNDR bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO underrun error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The LT_ERR_OVR and LT_ERR_UNDR bits are sticky and indicate, when set, that a line track error has occurred. These bits are valid for RX modes with recurring frame syncs only. If one of these bits is set, the programmed number of samples in PPI_COUNT did not match up with the actual number of samples counted between assertions of PPI_FS1 (for general-purpose modes) or start of active video (SAV) codes (for ITU-R 656 modes). If the PPI error interrupt is enabled in the SIC_IMASK register, an interrupt request is generated when one of these bits is set.

The LT_ERR_OVR flag signifies that a horizontal tracking overflow has occurred, where the value in PPI_COUNT was reached before a new SAV code was received. This flag does not apply for non ITU-R 656 modes; in this case, once the value in PPI_COUNT is reached, the PPI simply stops counting until receiving the next PPI_FS1 frame sync.

The LT_ERR_UNDR flag signifies that a horizontal tracking underflow has occurred, where a new SAV code or PPI_FS1 assertion occurred before the value in PPI_COUNT was reached.

PPI Delay Count Register (PPI_DELAY)

The PPI_DELAY register, shown in Figure 15-15, can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many PPI_CLK cycles to delay after assertion of PPI_FS1 before starting to read in or write out data.

Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.

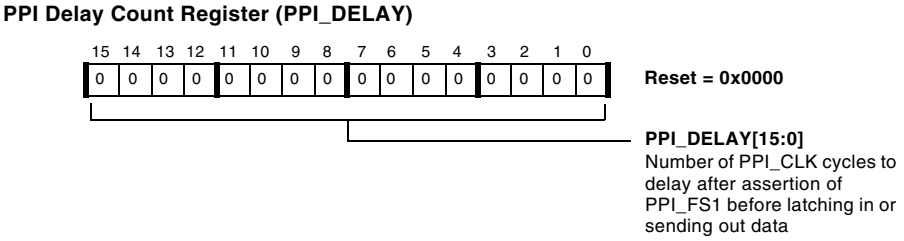

PPI Delay Count Register (PPI_DELAY)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PPI_DELAY[15:0]
Number of PPI_CLK cycles to delay after assertion of PPI_FS1 before latching in or sending out data

Figure 15-15. PPI Delay Count Register

PPI Transfer Count Register (PPI_COUNT)

The PPI_COUNT register, shown in Figure 15-16, is used in all modes except “RX mode with 0 frame syncs, external trigger” and “TX mode with 0 frame syncs.” For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the "horizontal" interval specified by PPI_FS1.

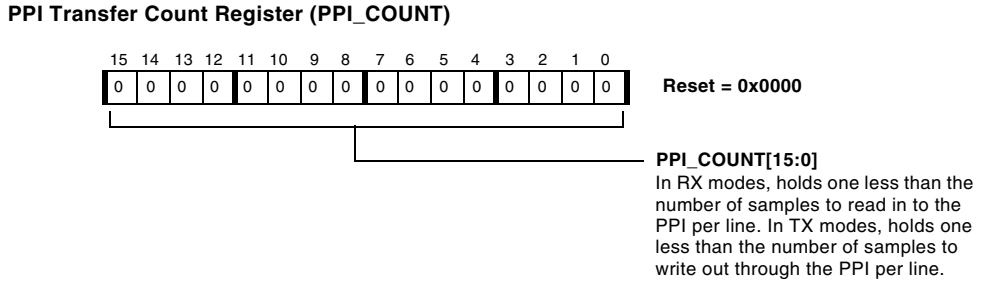

PPI Transfer Count Register (PPI_COUNT)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PPI_COUNT[15:0]
In RX modes, holds one less than the number of samples to read in to the PPI per line. In TX modes, holds one less than the number of samples to write out through the PPI per line.

Figure 15-16. PPI Transfer Count Register

PPI Lines Per Frame Register (PPI_FRAME)

The PPI_FRAME register, shown in Figure 15-17, is used in all TX and RX modes with two or three frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as field 1 and field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original "frame start" each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a frame track error results, and the FT_ERR bit in PPI_STATUS is set.

However, the PPI still automatically reinitializes to count to the value programmed in PPI_FRAME, and data transfer continues.

In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the field indicator. This occurs at the start of field 1.

In RX mode with three external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

When using RX mode with three external frame syncs, and only two syncs are needed, configure the PPI for 3-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.

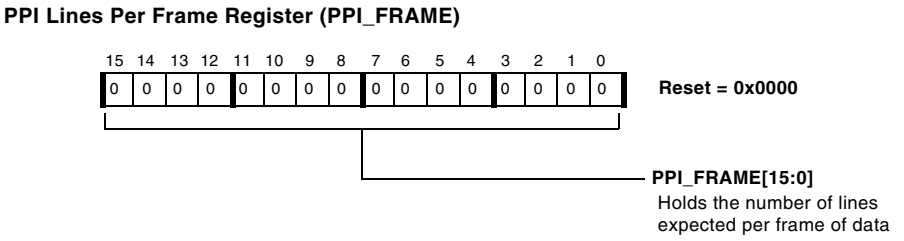

PPI Lines Per Frame Register (PPI_FRAME)
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
PPI_FRAME[15:0]
Holds the number of lines
expected per frame of data

Figure 15-17. PPI Lines Per Frame Register

Programming Examples

The PPI can be configured to receive data from a video source in several RX modes. The following programming examples (Listing 15-1 through Listing 15-5) describe the ITU-R 656 entire field input mode.

Listing 15-1. Configure DMA Registers

```
config_dma:
/*Assumes PPI is mapped to DMA channel 0.*/
  /* DMA0_START_ADDR */
  R0.L = rx_buffer;
  R0.H = rx_buffer;
  P0.L = lo(DMA0_START_ADDR);
  P0.H = hi(DMA0_START_ADDR);
  [P0] = R0;

  /* DMA0_CONFIG */
  R0.L = DI_EN | WNR;
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  W[P0] = R0.L;

  /* DMA0_X_COUNT */
  R0.L = 256;
  P0.L = lo(DMA0_X_COUNT);
  P0.H = hi(DMA0_X_COUNT);
  W[P0] = R0.L;

  /* DMA0_X_MODIFY */
  R0.L = 0x0001;
  P0.L = lo(DMA0_X_MODIFY);
  P0.H = hi(DMA0_X_MODIFY);
  W[P0] = R0.L;
  ssync;
config_dma.END:  RTS;
```

Listing 15-2. Configure PPI Registers

```
config_ppi:

 /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = 0x0004;
  W[P0] = R0.L;
  ssync;

config_ppi.END:  RTS;
```

Listing 15-3. Enable DMA

```
  /* DMA0_CONFIG */
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 15-4. Enable PPI

```
  /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 15-5. Clear DMA Completion Interrupt

```
/* DMA0_IRQ_STATUS */
P2.L = lo(DMA0_IRQ_STATUS);
P2.H = hi(DMA0_IRQ_STATUS);
R2.L = W[P2];
BITSET(R2,0);
W[P2] = R2.L;
ssync;
```

Unique Information for the ADSP-BF59x Processor

None.

16 SYSTEM RESET AND BOOTING

This document contains material that is subject to change without notice. The content of the boot ROM as well as hardware behavior may change across silicon revisions. See the anomaly list for differences between silicon revisions. This document describes functionality of silicon revision 0.0 of the ADSP-BF59x processors.

Overview

When the $\overline{\text{RESET}}$ input signal releases, the processor examines the state of the boot mode select pins (BMODE2-0) to determine the starting address for instruction execution. Based on the settings of these pins, instruction execution starts from either the base address of L1 ROM or the base address of Boot ROM.

The internal boot ROM includes a small boot kernel that loads application data from an external memory or host device. The application data is expected to be available in a well-defined format called the boot stream. A boot stream consists of multiple blocks of data and special commands that instruct the boot kernel how to initialize on-chip L1 memories as well as off-chip volatile memories.

The boot kernel processes the boot stream block-by-block until it is instructed by a special command to terminate the procedure and jump to the application's programmable start address, which traditionally is at 0xFFA0 0000 in on-chip L1 memory. This process is called "booting."

The processor features three dedicated input pins BMODE[2:0] that select the booting mode. The boot kernel evaluates the BMODE pins and performs booting from respective sources. Table 16-1 describes the modes of the BMODE pins.

Table 16-1. Booting Modes

BMODE2-0	Boot Source	Description
000	No boot – idle	The processor does not boot. Rather, the boot kernel executes an IDLE instruction.
001	Reserved	Reserved
010	Boot from external serial SPI memory using SPI1	After an initial device detection routine, the kernel boots from either 8-bit, 16-bit, 24-bit or 32-bit addressable SPI flash or EEPROM memory that connects to $\overline{\text{SPI1\_SSEL5}}$.
011	Boot from SPI host using SPI1	In this slave mode, the kernel expects the boot stream to be applied to SPI1 by an external host device.
100	Boot from external serial SPI memory using SPI0	After initial device detection routine, the kernel boots from either 8-bit, 16-bit, 24-bit, or 32-bit addressable SPI flash or EEPROM memory that connects to $\overline{\text{SPI0\_SSEL2}}$.
101	Boot from PPI host	In this boot mode, the kernel expects data to be received over the 16-bit PPI port. Data transfers are controlled with the incoming PPI_FS1 signal. The processor uses the PPI_FS2 signal to indicate when it is ready to receive data and how much data is expected.
110	Boot from UART0 host	In this slave mode, the kernel expects the boot stream to be applied to UART0 by an external host device. Prior to providing the boot stream, the host device is expected to send a 0x40 (ASCII '@') character that is examined by the kernel to adjust the bit rate.
111	Boot from internal L1 ROM (full user control)	In this mode, the processor starts instruction execution at the base address of on-chip L1 instruction ROM, entirely bypassing the boot ROM.

Reset and Power-up

Table 16-2 describes the six types of resets.

Note that each type resets the core except for the System Software reset.

Table 16-2. Resets

Reset	Source	Result
Hardware reset	The RESET pin causes a hardware reset.	Resets both the core and the peripherals, including the dynamic power management controller (DPMC). Resets bits [15:4] of the SYSCR register. For more information, see "System Reset Configuration (SYSCR) Register" on page 16-54.
Power-on reset	Internal circuitry recognizes initial power-up condition.	Resets all registers (core and system) provided that the power up guidelines are followed for the RESET pin.
System software reset	Calling the bfrom_SysControl() routine with the SYSCTRL_SYSRESET option triggers a system reset.	Resets only the peripherals, excluding most of the DPMC. The system software reset clears bits [15:13] and bits [11:4] of the SYSCR register, but not the WURESET bit. The core is not reset and a boot sequence is not triggered. Sequencing continues at the instruction after bfrom_SysControl() returns.
Watchdog timer reset	Programming the watchdog timer causes a watchdog timer reset.	Resets both the core and the peripherals, excluding most of the DPMC. (Because of the partial reset to the DPMC, the watchdog timer reset is not functional when the processor is in Sleep or Deep Sleep modes.) The SWRST or the SYSCR register can be read to determine whether the reset source was the watchdog timer.

Table 16-2. Resets (Continued)

Reset	Source	Result
Core double-fault reset	A core double fault occurs when an exception happens while the exception handler is executing. If the core enters a double-fault state, and the Core Double Fault Reset Enable bit (DOUBLE_FAULT) is set in the SWRST register, then a software reset will occur.	Resets both the core and the peripherals, excluding most of the DPMC. The SWRST or SYSCR registers can be read to determine whether the reset source was a core double-fault.
Core-only software reset	This reset is caused by executing a RAISE 1 instruction or by setting the software reset (SYSRST) bit in the core debug control register (DBGCTL) through emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Resets all core registers. Program execution vectors to the 0xEF00 0000 address. The boot code executes an immediate system reset to ensure system consistency.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted after a specified asserted hold time to perform a hardware reset. For more information, see the product data sheet.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the $\overline{\text{RESET}}$ pin is deasserted, the processor ensures that all asynchronous peripherals have recognized and completed a reset. After the reset, the processor transitions into the boot mode sequence configured by the state of the BMODE pins.

The BMODE pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either V_{DDEXT} or GND. The pins and the corresponding bits in the SYSCR register configure the boot mode that is employed after hardware reset or system software reset. See the *Blackfin Processor Programming Reference* for further information.

Software Resets

A software reset may be initiated in three ways.

- By the watchdog timer, if appropriately configured
- Calling the `bfrom_SysControl()` API function residing in the on-chip ROM. For further information, see Chapter 6, "Dynamic Power Management".
- By the `RAISE 1` instruction

The watchdog timer resets both the core and the peripherals, as long as the processor is in Active or Full-On mode. A system software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

In order to perform a system reset, the `bfrom_SysControl()` routine must be called while executing from L1 memory.

After either the watchdog or system software reset is initiated, the processor ensures that all asynchronous peripherals have recognized and completed a reset.

For a reset generated by formatting the watchdog timer, the processor transitions into the boot mode sequence. The boot mode is configured by the state of the `BMODE` bit field in the `SYSCR` register.

A software reset is initiated by executing the `RAISE 1` instruction or setting the software reset (`SYSRST`) bit in the core debug control register (`DBGCTL`) (`DBGCTL` is not visible to the memory map) through emulation software through the JTAG port.

A software reset only affects the state of the core. The boot kernel immediately issues a system reset to keep consistency with the system domain.

Reset Vector

When reset releases, the processor starts fetching and executing instructions from *either* address 0xEF00 0000 *or* 0xFFA1 0000, based on the settings of boot mode select pins.

On a hardware reset, the boot kernel initializes the EVT1 register to 0xFFA0 0000. When the booting process completes, the boot kernel jumps to the location provided by the EVT1 vector register. The content of the EVT1 register is overwritten by the TARGET ADDRESS field of the first block of the applied boot stream. If the BCODE field of the SYSCR register is set to 3 (no boot option), the EVT1 register is not modified by the boot kernel on software resets. Therefore, programs can control the reset vector for software resets through the EVT1 register. This process is illustrated by the flow chart in Figure 16-1.

The content of the EVT1 register may be undefined in emulator sessions.

Servicing Reset Interrupts

The processor services a reset event like other interrupts. The reset interrupt has top priority. Only emulation events have higher priority. When coming out of reset, the processor is in supervisor mode and has full access to all system resources. The boot kernel can be seen as part of the reset service routine. It runs at the top interrupt priority level.

Even when the boot process has finished and the boot kernel passes control to the user application, the processor is still in the reset interrupt. To enter user mode, the reset service routine must initialize the RETI register and terminate with an RTI instruction.

For a programming example, see “System Reset” on page 16-71.

Listing 16-1 and Listing 16-2 on page 16-72 show code examples that handle the reset event. See the Blackfin Processor Programming Reference for details on user and supervisor modes.

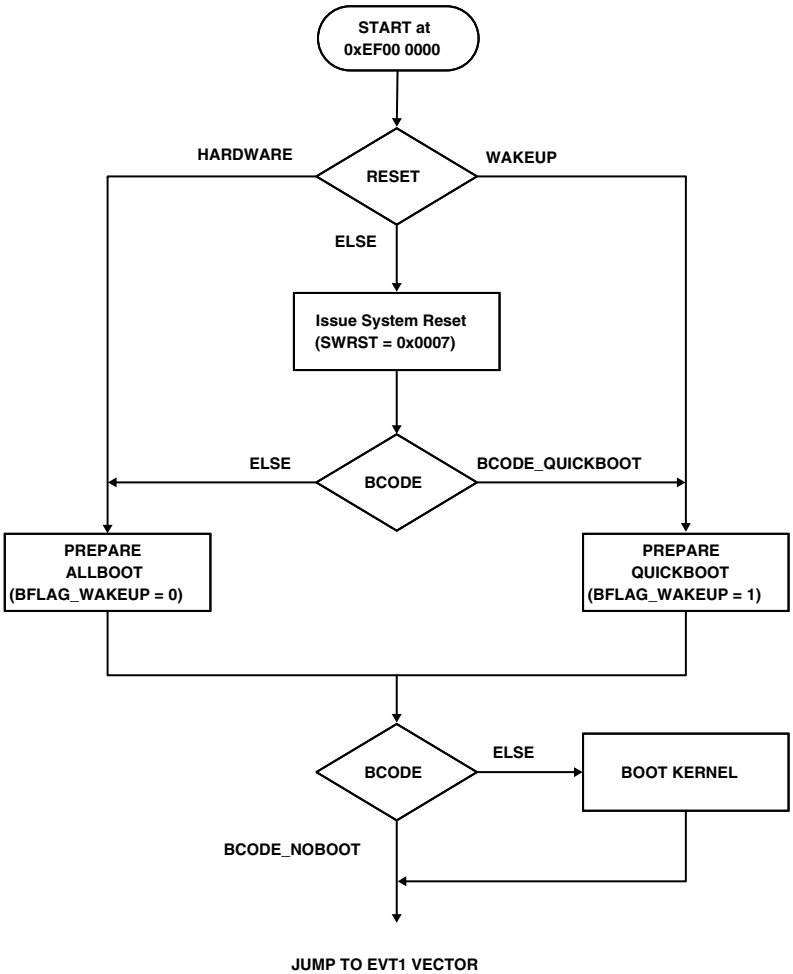


Figure 16-1. Global Boot Flow

Systems that do not work in an operating system environment may not enter user mode. Typically, the interrupt level needs to be degraded down to IVG15. Listing 16-3 and Listing 16-4 on page 16-73 show how this is accomplished.

Since the boot kernel is running at reset interrupt priority, NMI events, hardware errors and exceptions are not serviced at boot time. As soon as the reset service routine returns, the processor can service the events that occurred during the boot sequence. It is recommended that programs install NMI, hardware error, and exception handlers before leaving the reset service routine. This includes proper initialization of the respective event vector registers `EVTx`.

Basic Booting Process

After evaluating the `BMODE` pins, the boot kernel residing in the on-chip boot ROM starts processing the boot stream. The boot stream is either read from memory or received from a host processor. A boot stream represents the application data and is formatted in a special manner. The application data is segmented into multiple blocks of data. Each block begins with a block header. The header contains control words such as the destination address and data length information.

As Figure 16-2 on page 16-9 illustrates, the VisualDSP++ tools suite features a loader utility (`elfloader.exe`). The loader utility parses the input executable file (`.DXE`), segments the application data into multiple blocks, and creates the header information for each block. The output is stored in a loader file (`.LDR`). The loader file contains the boot stream and is made available to hardware by programming or burning it into non-volatile external memory. Refer to the *VisualDSP++ Loader Manual* for information on switches for loader files.

Figure 16-3 on page 16-10 shows the parallel or serial boot stream contained in a flash memory device. In host boot scenarios, the non-volatile memory more likely connects to the host processor rather than directly to the Blackfin processor. After reset, the headers are read and parsed by the

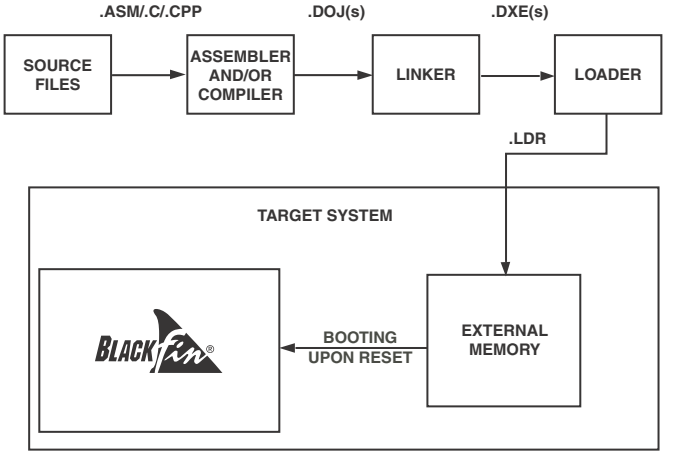


Figure 16-2. Project Flow for a Standalone System

on-chip boot ROM, and processed block-by-block. Payload data is copied to destination addresses in on-chip L1 memory.

Booting into scratchpad memory (0xFFB0 0000–0xFFB0 0FFF) is not supported. If booting to scratchpad memory is attempted, the processor hangs within the on-chip boot ROM. Similarly, booting into the upper 16 bytes of L1 data bank B (0xFF80 7FF0–0xFF80 7FFF by default) is not supported. These memory locations are used by the boot kernel for intermediate storage of block header information. These memory regions cannot be initialized at boot time. After booting, they can be used by the application during runtime.

When the `BFLAG_INDIRECT` flag for any block is set, the boot kernel uses another memory block in L1 data bank B (by default, 0xFF80 7F00–0xFF80 7FEF) for intermediate data storage. To avoid conflicts, the VisualDSP++ `elfloader` utility ensures this region is booted last.

The entire source code of the boot ROM is shipped with the VisualDSP++ tools installation. Refer to the source code for any additional questions not covered in this manual. Note that minor maintenance

work may be done to the content of the boot ROM when silicon is updated.

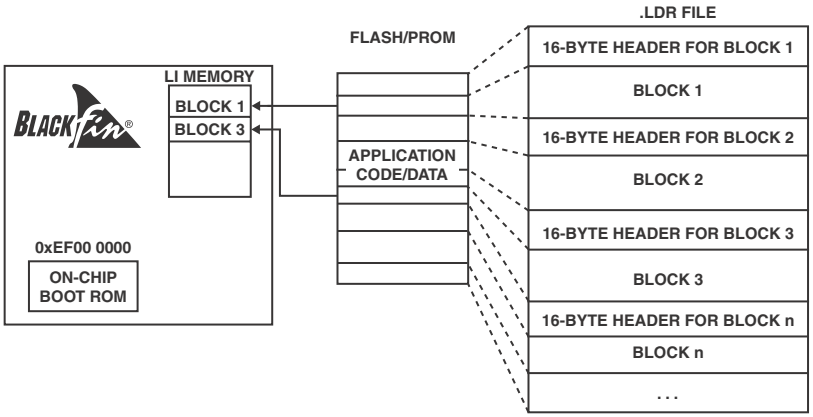


Figure 16-3. Booting Process

Block Headers

A boot stream consists of multiple boot blocks, as shown in Figure 16-3 on page 16-10. Every block is headed by a 16-byte block header. However, every block does not necessarily have a payload, as shown in Figure 16-4 on page 16-11.

The 16 bytes of the block header are functionally grouped into four 32-bit words, the `BLOCK CODE`, the `TARGET ADDRESS`, the `BYTE COUNT`, and the `ARGUMENT` fields.

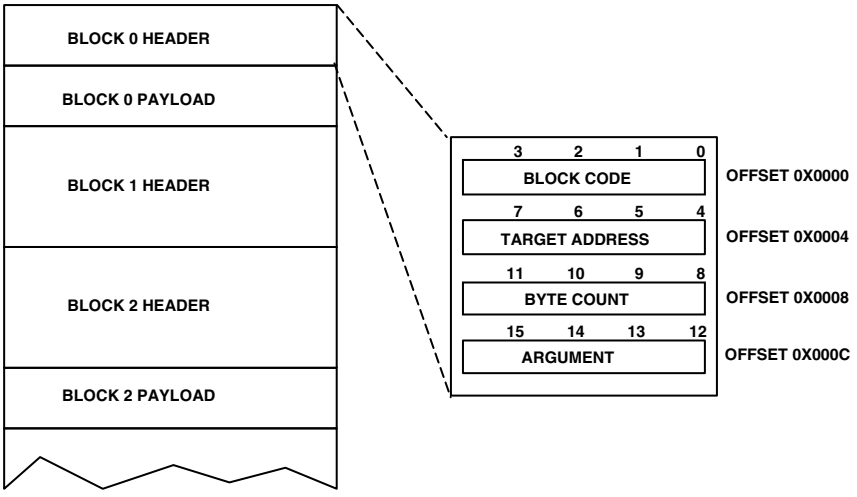


Figure 16-4. Boot Stream Headers

Block Code

The first 32-bit word is the BLOCK CODE. See Figure 16-5.

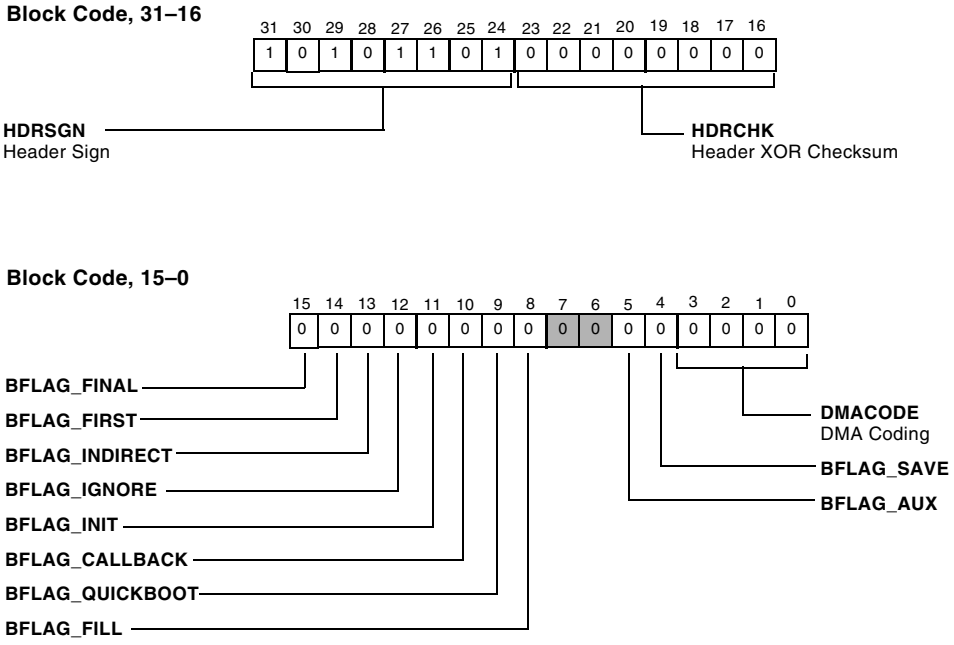

Block Code, 31–16
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
1 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0
HDRSGN
Header Sign
HDRCHK
Header XOR Checksum
Block Code, 15–0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BFLAG_FINAL
BFLAG_FIRST
BFLAG_INDIRECT
BFLAG_IGNORE
BFLAG_INIT
BFLAG_CALLBACK
BFLAG_QUICKBOOT
BFLAG_FILL
DMACODE
DMA Coding
BFLAG_SAVE
BFLAG_AUX

Figure 16-5. Block Code, 31–0

DMA Code Field

The DMA code (DMACODE) field instructs the boot kernel whether to use 8-bit, 16-bit or 32-bit DMA and how to program the source modifier of a memory DMA. Particularly in case of memory boot modes, this field is interrogated by the boot kernel to differentiate the 8-bit, 16-bit, and 32-bit cases.

The boot kernel tests this field only on the first block and ignores the field in further blocks (See Table 16-3).

Table 16-3. Bus and DMA Width Coding

DMA Code	DMA Width	Source DMA Modify	Application
0	reserved[1]		
1	8-bit	1	Default 8-bit boot from 8-bit source[2]
2	8-bit	2	reserved[3]
3	8-bit	4	reserved[3]
4	8-bit	8	reserved[3]
5	8-bit	16	reserved[3]
6	16-bit	2	Default 16-bit boot from 16-bit source
7	16-bit	4	reserved[3]
8	16-bit	8	reserved[3]
9	16-bit	16	reserved[3]
10	32-bit	4	Default 32-bit boot from 32-bit source[4]
11	32-bit	8	reserved[3]
12	32-bit	16	reserved[3]
13	64-bit	8	Default 64-bit boot from 64-bit source[3]
14	64-bit	16	reserved[3]
15	128-bit	16	Default 128-bit boot from 128-bit source[3]

1 Reserved to differentiate from ADSP-BF53x boot streams.
2 Used by all byte-wise serial boot modes.
3 Not supported by ADSP-BF59x Blackfin products.
4 Applicable only to memory boot modes.

Block Flags Field

Table 16-4. Block Flags

Bit	Name	Description
4	BFLAG_SAVE	Saves the memory of this block to off-chip memory in case of power failure or a hibernate request. This flag is not used by the on-chip boot kernel.
5	BFLAG_AUX	Nests special block types as required by special purpose second-stage loaders. This flag is not used by the on-chip boot kernel.
6	Reserved	
7	Reserved	
8	BFLAG_FILL	Tells the boot kernel to not process any payload data. Instead the target memory (specified by the `TARGET ADDRESS` and `BYTE COUNT` fields) is filled with the 32-bit value provided by the `ARGUMENT` word. The fill operation is always performed by 32-bit DMA; therefore target address and byte count must be divisible by four.
9	BFLAG_QUICKBOOT	Processes the block for full boot only. Does not process this block for a quick boot (warm boot).
10	BFLAG_CALLBACK	Calls a subfunction that may reside in on-chip or off-chip ROM or is loaded by an initcode in advance. Often used with the `BFLAG_INDIRECT` switch. If `BFLAG_CALLBACK` is set for any block, an initcode must register the callback function first. The function is called when either the entire block is loaded or the intermediate storage memory is full. The callback function can do advanced processing such as CRC checksum.
11	BFLAG_INIT	This flag causes the boot kernel to issue a `CALL` instruction to the target address of the boot block after the entire block is loaded. The initcode should return by an `RTS` instruction. It may or may not be overwritten by application data later in the boot process. If the code is loaded earlier or resides in ROM, the init block can be zero sized (no payload).

Table 16-4. Block Flags (Continued)

Bit	Name	Description
12	BFLAG_IGNORE	Indicates a block that is not booted into memory. It instructs the boot kernel to skip the number of bytes of the boot stream as specified by BYTE COUNT. In master boot modes, the boot kernel simply modifies its source address pointer. In this case the BYTE COUNT value can be seen as a 32-bit two's-complement offset value to be added to the source address pointer. In slave boot modes, the boot kernel actively loads and changes the payload of the block. In slave modes the byte count must be a positive value.
13	BFLAG_INDIRECT	Boots to an intermediate storage place, allowing for calling an optional callback function, before booting to the destination. This flag is used when the boot source does not have DMA support and either the destination cannot be accessed by the core (L1 instruction SRAM) or cannot be efficiently accessed by the core. This flag is also used when CALLBACK requires access to data to calculate a checksum, or when performing tasks such as decryption or decompression.
14	BFLAG_FIRST	This flag, which is only set on the first block of a DXE, tells the boot kernel about the special nature of the TARGET ADDRESS and the ARGUMENT fields. The TARGET ADDRESS field holds the start address of the application. The ARGUMENT field holds the offset to the next DXE.
15	BFLAG_FINAL	This flag causes the boot kernel to pass control over to the application after the final block is processed. This flag is usually set on the last block of a DXE unless multiple DXEs are merged.

The BFLAG_FIRST flag must not be combined with the BFLAG_FILL flag. The BFLAG_FIRST flag may be combined with the BFLAG_IGNORE flag to deposit special user data at the top of the boot stream. Note the special importance of the VisualDSP++ elfloader -readall switch.

Header Checksum Field

The header checksum (HDRCHK) field holds a simple XOR checksum of the other 31 bytes in the boot block header. The boot kernel jumps to the error routine if the result of an XOR operation across all 32 header bytes (including the HDRCHK value) differs from zero. The default error routine is

a simple IDLE; instruction. The user can overwrite the default error handler using the initcode mechanism.

Header Sign Field

The header signature (HDRSGN) byte always reads as 0xAD and is used to verify whether the block pointer actually points to a valid block.

Target Address

This 32-bit field holds the target address where the boot kernel loads the block payload data. When the BFLAG_FILL flag is set, the boot kernel fills the memory with the value stored in the ARGUMENT field starting at this address. If the BFLAG_INIT flag is set the kernel issues a CALL(TARGET ADDRESS) instruction after the optional payload is loaded.

If the BFLAG_FIRST flag is set, the TARGET ADDRESS field contains the start address of the application to which the boot kernel jumps at the end of the boot process. This address will also be stored in the EVT1 register. By default the VisualDSP++ elfloader utility sets this value to 0xFFA0 0000 for compatibility with other Blackfin products.

The target address should be divisible by four, because the boot kernel uses 32-bit DMA for certain operations. The target address must point to valid on-chip memory locations. When booting through peripherals that do not support DMA transfers, the BFLAG_INDIRECT flag must be set if the target address points to L1 instruction memory.

Booting to scratchpad memory is not supported. The scratchpad memory functions as a stack for the boot kernel. The L1 data memory locations 0xFF80 7FF0 to 0xFF80 7FFF are used by the boot kernel and should not be overwritten by the application. The memory range used for intermediate storage as controlled by the BFLAG_INDIRECT switch should only be booted after the last BFLAG_INDIRECT bit is processed. By default the address range 0xFF80 7F00–0xFF80 7FEF is used for intermediate storage.

For normal boot operation, the target address points to RAM memory. There are however a few exceptions where the target address can point to on-chip or off-chip ROM. For example a zero-sized BFLAG_INIT block would instruct the boot kernel to call a subroutine residing in ROM or flash memory. This method is used to activate the CRC32 feature.

Byte Count

This 32-bit field tells the boot kernel how many bytes to process. Normally, this is the size of the payload data of a boot block. If the BFLAG_FILL flag is set there is no payload. In this case the BYTE COUNT field uses the value in its ARGUMENT field to tell the boot kernel how many bytes to process.

The byte count is a 32-bit value that should be divisible by four. Zero values are allowed in all block types. Most boot modes are based upon DMA operation which are only 16-bit words for Blackfin processors. The boot kernel may therefore start multiple DMA work units for large boot blocks. This enables a single block to fill to zero the memory, for example, resulting in compact boot streams. The HWAIT signal may toggle for each work unit.

If the BFLAG_IGNORE flag is set, the byte count is used to redirect the boot source pointer to another memory location. In master boot modes, the byte count is a two's-complement (signed long integer) value. In slave boot modes, the value must be positive.

Argument

This 32-bit field is a user variable for most block types. The value is accessible by the initcode or the callback routine and can therefore be used for optional instructions to these routines. When the CRC32 feature is activated, the ARGUMENT field holds the checksum over the payload of the block.

When the BFLAG_FILL flag is set there is no payload. The argument contains the 32-bit fill value, which is most likely a zero.

If the BFLAG_FIRST flag is set, the argument contains the relative next-DXE pointer for multi-DXE applications. For single-DXE applications the field points to the next free boot source address after the current DXE's boot stream.

Boot Host Wait (HWAIT) Feedback Strobe

The HWAIT feedback strobe is a handshake signal that is used to hold off the host device from sending further data while the boot kernel is busy.

On ADSP-BF59x processors, this feature is implemented by a GPIO that is toggled by the boot kernel as required. The PG4 GPIO is used for this purpose.

The signal polarity of the HWAIT strobe is programmable by an external resistor in the 10 kΩ range.

A pull-up resistor instructs the HWAIT signal to be active high. In this case the host is permitted to send header and footer data when HWAIT is low, but should pause while HWAIT is high. This is the mode used in SPI slave boot on other Blackfin products.

Similarly, a pull-down resistor programs active-low behavior.

Note that the HWAIT signal is implemented slightly differently than on ADSP-BF53x Blackfin processors. In the ADSP-BF59x processors, the meaning of the pulling resistor is inverted and HWAIT is asserted by default during reset.

The boot kernel first senses the polarity on the respective HWAIT pin. Then it enables the output driver but keeps the signal in its asserted state. The signal is not released until the boot kernel is ready for data, or when a receive DMA is started. As soon as the DMA completes, HWAIT becomes active again.

The boot host wait signal holds the host from booting in any slave boot mode and prevents it from being overrun with data. The HWAIT signal is, however, available in all boot modes.

In general the host device must interrogate the HWAIT signal before every word that is sent. This requirement can be relaxed for boot modes using on-chip peripherals that feature larger receive FIFOs. However, the host must not rely on the DMA FIFO since its content is cleared at the end of a DMA work unit.

While the HWAIT signal is only used for boot purposes, it may also play a significant role after booting. In slave boot modes, for example, the host device does not necessarily know whether the Blackfin processor is in an active mode or a power-down mode. For example, the HWAIT signal can be used to signal when the processor is in hibernate mode.

Using HWAIT as Reset Indicator

While the HWAIT signal is mandatory in some boot modes, it is optional in others.

If using a pull-up resistor, the HWAIT signal is driven low for the rest of the boot process (and beyond). If using a pull-down resistor, HWAIT is driven high.

With a pull-down resistor, this feature can be used to simulate an active-low reset output. When the processor is reset, or in hibernate, the GPIO is in a high impedance state and HWAIT is pulled low by the resistor. As soon as the processor recovers and has settled the PLL again, the HWAIT is driven high and can alert external circuits.

Boot Termination

After the successful download of the application into the bootable memory, the boot kernel passes control to the user application. By default this is performed by jumping to the vector stored in the EVT1 register. The

boot kernel provides options to execute an RTS instruction or a RAISE 1 instruction instead. The default behavior can be changed by an initcode routine. The EVT1 register is updated by the boot kernel when processing the BFLAG_FIRST block. See "Servicing Reset Interrupts" on page 16-6 to learn how the application can take control.

Before the boot kernel passes program control to the application it does some housekeeping. Most of the registers that were used are changed back to their default state but some register values may differ for individual boot modes. DMA configuration registers and primary register control registers (UARTx_LCR, SPIx_CTL, etc.) are restored, while others are purposely not restored. For example SPIx_BAUD, UARTx_DLH and UARTx_DLL remain unchanged so that settings obtained during the booting process are not lost.

Single Block Boot Streams

The simplest boot stream consists of a single block header and one contiguous block of instructions. With appropriate flag instructions the boot kernel loads the block to the target address and immediately terminates by executing the loaded block.

Table 16-5 shows an example of a single block boot stream header that could be loaded from any serial boot mode. It places a 256-byte block of instructions at L1 instruction SRAM address 0xFFA0 0000. The flags BFLAG_FIRST and BFLAG_FINAL are both set at the same time. Advanced flags, such as BFLAG_IGNORE, BFLAG_INIT, BFLAG_CALLBACK and BFLAG_FILL, do not make sense in this context and should not be used.

Table 16-5. Header for a Single Block Boot Stream

Field	Value	Comments
BLOCK CODE	0xAD33 C001	0xAD00 0000 | XORSUM | BFLAG_FINAL | BFLAG_FIRST | (DMACODE & 0x1)
TARGET ADDRESS	0xFFA0 0000	Start address of block and application code

Table 16-5. Header for a Single Block Boot Stream (Continued)

Field	Value	Comments
BYTE COUNT	0x0000 0100	256 bytes of code
ARGUMENT	0x0000 0100	Functions as next-DXE pointer in multi-DXE boot streams

With the `BFLAG_FIRST` flag set, the `ARGUMENT` field functions as the next-DXE pointer. This is a relative pointer to the next free source address or to the next DXE start address in a multi-DXE stream.

Advanced Boot Techniques

The following sections describe advanced boot techniques. These techniques are useful for customers developing custom boot routines.

Initialization Code

Initcode routines are subroutines that the boot kernel calls during the booting process. The user can customize and speed up the booting mechanisms using this feature. Traditionally, an initcode is used to set up system PLL, bit rates, and other system settings. If executed early in the boot process, the boot time can be significantly reduced.

After the payload data is loaded for a specific boot block, if the `BFLAG_INIT` flag is set, the boot kernel issues a `CALL` instruction to the target address of the block.

On ADSP-BF59x Blackfin processors, initcode routines follow the C language calling convention so they can be coded in C language or assembly.

The expected prototype is:

```
void initcode(ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the ADI_BOOT_INITCODE_FUNC type:

```
typedef void ADI_BOOT_INITCODE_FUNC (ADI_BOOT_DATA* ) ;
```

Optionally, the initcode routine can interrogate the formatting structure and customize its own behavior or even manipulate the regular boot process. A pointer to the structure is passed in the R0 register. Assembly coders must ensure that the routine returns to the boot kernel by a terminating RTS instruction.

Initcodes can rely on the validity of the stack, which resides in scratchpad memory. The ADI_BOOT_DATA structure resides on the stack. Rules for register usage conform to the compiler conventions. See the *VisualDSP++ C/C++ Compiler and Library Manual* for more information.

In the simple case, initcodes consist of a single instruction section and are represented by a single block within the boot stream. This block has the BFLAG_INIT bit set.

An init block can consist of multiple sections where multiple boot blocks represent the initcode within the boot stream. Only the last block has the BFLAG_INIT bit set.

The VisualDSP++ elfloader utility ensures that the last of these blocks vectors to the initcode entry address. The utility instructs the on-chip boot ROM to execute a CALL instruction to the given target address.

When the on-chip boot ROM detects a block with the BFLAG_INIT bit set, it boots the block into Blackfin memory and then executes it by issuing a CALL to its target address. For this reason, every initcode must be terminated by an RTS instruction to ensure that the processor vectors back to the on-chip boot ROM for the rest of the boot process.

Sometimes initcode boot blocks have no payload and the BYTE COUNT field is set to zero. Then the only purpose of the block may be to instruct the boot kernel to issue the CALL instruction.

Initcode routines can be very different in nature. They might reside in ROM or SRAM. They might be called once during the booting process or multiple times. They might be volatile and be overwritten by other boot blocks after executing, or they might be permanently available after boot time. The boot kernel has no knowledge of the nature of initcodes and has no restrictions in this regard. Refer to the *VisualDSP++ Loader and Utilities Manual* for how this feature is supported by the tools chain.

It is the user's responsibility to ensure that all code and data sections that are required by the initcode are present in memory by the time the initcode executes. Special attention is required if initcodes are written in C or C++ language. Ensure that the initcode does not contain calls to the runtime libraries. Do not assume that parts of the runtime environment, such as the heap are fully functional. Ensure that all runtime components are loaded and initialized before the initcode executes.

The VisualDSP++ elfloader utility provides two different mechanisms to support the initcode feature.

- The `-init initcode.dxe` command line switch
- The `-initcall address/symbol` command line switch

If enabled by the VisualDSP++ elfloader `-init initcode.DXE` command line switch, the initcode is added to the beginning of the boot stream. Here, `initcode.DXE` refers to the user-provided custom initialization executable— a separate VisualDSP++ project. Figure 16-6 on page 16-24 shows a boot stream example that performs the following steps.

1. Boot initcode into L1 memory.
2. Execute initcode.
3. Overwrite initcode with final application code.
4. Boot data/code into memory.
5. Continue program execution with block n.

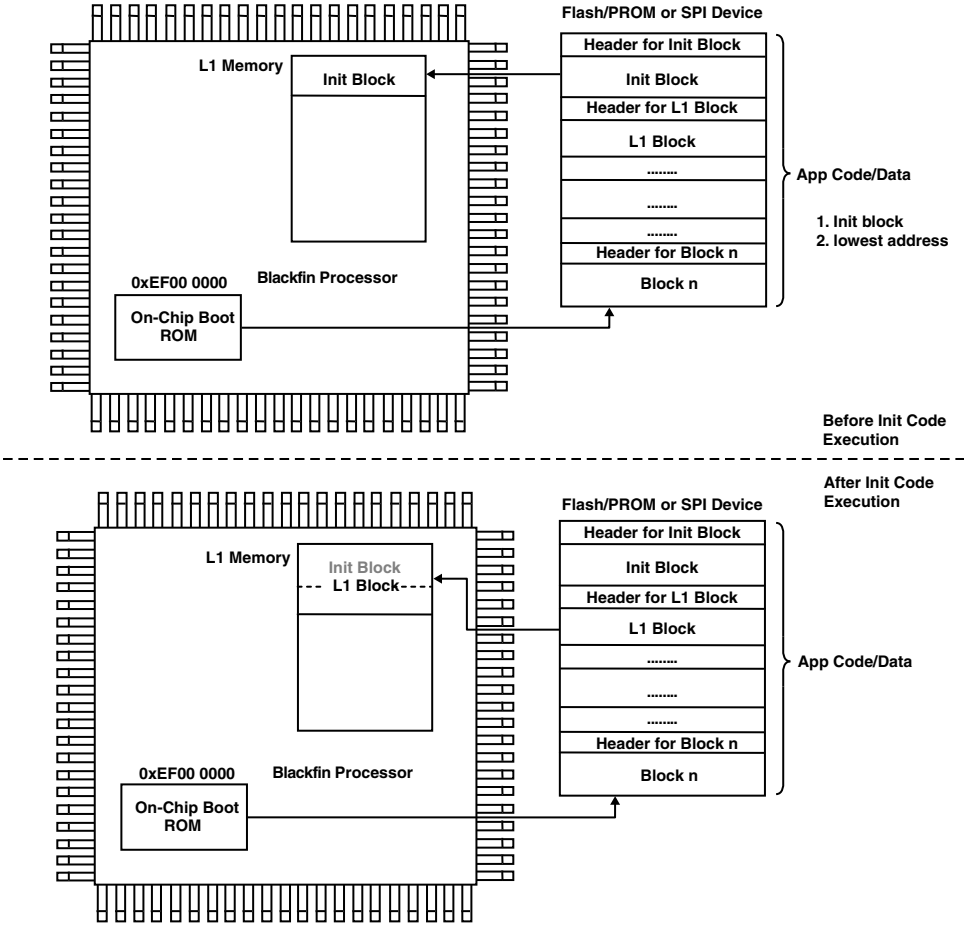


Figure 16-6. Initialization Code Execution/Boot

Although `initcode.DXE` files are built as VisualDSP++ projects, they differ from standard projects. Initcodes provide only a callable sub-function, so they look more like a library than an application. Nevertheless, unlike library files (`.DLB` file extension), the symbol addresses have already been resolved by the linker.

An initcode is always a heading for the regular application code. Consequently whether the initcode consists of one or multiple blocks, it is not terminated by a BFLAG_FINAL bit indicator—this would cause the boot ROM to terminate the boot process.

It is advantageous to have a clear separation between the initcode and the application by using the -init switch. If this separation is not needed, the elfloader -initcall command-line switch might be preferred. It enables fractions of the application code to be traded as initcode during the boot process. See the *VisualDSP++ Loader and Utilities Manual* for further details.

Initcode examples are shown in "Programming Examples" on page 16-71.

Quick Boot

In some booting scenarios, not all memories need to be re-initialized.

The ADSP-BF59x processor's boot kernel can conditionally process boot blocks. The normal scenario is all boot, the shortened version is quick boot. It relies on the following primitives.

- The SYSCR register is read to determine what kind of boot is expected from the boot kernel. Refer to Figure 16-20 on page 16-54.

  The WURESET bit is used to distinguish between cold boot and warm boot situations and to identify wake-up from hibernate situations.

  The BCODE bit field in the SYSCR register can overrule the native decision of the boot kernel for a software boot. See the flowchart in Figure 16-1 on page 16-7.

- The BFLAG_WAKEUP bit in the dFlag word of the ADI_BOOT_DATA structure indicates that the final decision was to perform a quick boot. If the boot kernel is called from the application, then the

application can control the boot kernel behavior by setting the BFLAG_WAKEUP flag accordingly. See the dFlags variable on Figure 16-25 on page 16-63.

- The BFLAG_QUICKBOOT flag in the BLOCK CODE word of the block header controls whether the current block is ignored for quick boot.

If both the global BFLAG_WAKEUP and the block-specific BFLAG_QUICKBOOT flags are set, the boot kernel ignores those blocks. But since the BFLAG_INIT, BFLAG_CALLBACK, BFLAG_FINAL, and BFLAG_AUX flags are internally cleared and the BFLAG_IGNORE flag is toggled, through double negation, the "ignore the ignore block" command instructs the boot kernel to process the block.

Although the BFLAG_INIT flag is suppressed in quick boot, the user may not want to combine the BFLAG_INIT flag with the BFLAG_QUICKBOOT flag. The initialization code can interrogate the BFLAG_WAKEUP flag and execute conditional instructions.

Indirect Booting

The processor's boot kernel provides a control mechanism to let blocks either boot directly to their final destination or load to an intermediate storage place, then copy the data to the final destination in a second step.

This feature is motivated by the following requirements:

- Some boot modes do not use DMA. They load data by core instruction. The core cannot access some memories directly (for example L1 instruction SRAM), or is less efficient than the DMA in accessing some memories.
- In some advanced booting scenarios, the core needs to access the boot data during the booting process, for example in processing decompression, decryption and checksum algorithms at boot time.

The indirect booting option helps speed-up and simplify such scenarios. Software accesses off-chip memory less efficiently and cannot access data directly if it resides in L1 instruction SRAM.

Indirect booting is not a global setting. Every boot block can control its own processing by the BFLAG_INDIRECT flag in the block header.

In general a boot block may not fit into the temporary storage memory so the boot kernel processes the block in multiple steps. The larger the temporary buffer, the faster the boot process. By default the L1 data memory region between 0xFF80 7F00 and 0xFF80 7FEF is used for intermediate storage. Initialization code can alter this region by modifying the pTempBuffer and dTempByteCount variables in the ADI_BOOT_DATA structure. The default region is at the upper end of a physical memory block. When increasing the dTempByteCount value, pTempBuffer also has to change.

Callback Routines

Callback routines, like initialization codes, are user-defined subroutines called by the boot kernel at boot time. The BFLAG_CALLBACK flag in the block header controls whether the callback routine is called for a specific block.

There are several differences between initcodes and callback routines. While the BFLAG_INIT flag causes the boot kernel to issue a CALL instruction to the target address of the specific boot block, the BFLAG_CALLBACK flag causes the boot kernel to issue a CALL instruction to the address held by the pCallBackFunction pointer in the ADI_BOOT_DATA structure. While a boot stream can have multiple individual initcodes, it can have just one callback routine. In the standard boot scenario, the callback routine has to be registered by an initcode prior to the first block that has the BFLAG_CALLBACK flag set.

The purpose of the callback routine is to apply standard processing to the block data. Typically, callback routines contain checksum, decryption,

decompression, or hash algorithms. Checksum or hash words can be passed through the block header ARGUMENT field.

Since callback routines require access to the payload data of the boot blocks, the block data must be loaded before it can be processed. Unlike initcodes, a callback usually resides permanently in memory. If the block is loaded to L1 instruction memory, the BFLAG_CALLBACK flag is likely combined with the BFLAG_INDIRECT bit. The boot kernel performs these steps in the following order.

1. Data is loaded into the temporary buffer defined by the pTempBuffer variable.
2. The CALL to the pCallBackFunction is issued.
3. After the callback routine returns, the memory DMA copies data to the destination.

If a block does not fit into the temporary buffer, for example when the BLOCK COUNT is greater than the dTempByteCount variable, the three steps are executed multiple times until all payload data is loaded and processed. The boot kernel passes the parameter dCbFlags to the callback routine to tell it that it is being invoked the first or the last time for a specific block. To store intermediate results across multiple calls the callback routine can use the uwUserShort and dUserLong variables in the ADI_BOOT_DATA structure.

Callback routines meet C language calling conventions for subroutines. The prototype is as follows.

```
s32 CallBackFunction (ADI_BOOT_DATA* pBootStruct,
ADI_BOOT_BUFFER* pCallbackStruct, s32 dCbFlags);
```

The VisualDSP++ header file defines the ADI_BOOT_CALLBACK_FUNC type the following way:

```
typedef s32 ADI_BOOT_CALLBACK_FUNC (ADI_BOOT_DATA*,
ADI_BOOT_BUFFER*, s32 ) ;
```

The pBootStruct argument is passed in R0 and points to the ADI_BOOT_DATA structure used by the boot kernel. These are handled by the pTempBuffer and dTempByteCount variables as well as the pHeader pointer to the ARGUMENT field. The callback routine may process the block further by modifying the pTempBuffer and dTempByteCount variables.

The pCallbackStruct structure passed in R1 provides the address and length of the data buffer. When the BFLAG_INDIRECT flag is not set, the pCallbackStruct contains the target address and byte count of the boot block. If the BFLAG_INDIRECT flag is set, the pCallbackStruct contains a copy of the pTempBuffer. Depending on the size of the boot block and processing progress, the byte count provided by pCallbackStruct equals either dTempByteCount or the remainder of the byte count.

When the BFLAG_INDIRECT flag is set along with the BFLAG_CALLBACK flag, memory DMA is invoked by the boot kernel after the callback routine returns. This memory DMA relies on the pCallbackStruct structure not the global pTempBuffer and dTempByteCount variables.

The callback routine can control the source of the memory DMA by altering the content of the pCallbackStruct structure, as may be required if the callback routine performs data manipulation such as decompression.

The dCbFlags parameter passed in R2 tells the callback routine whether it is invoked the first time (CBFLAG_FIRST) or whether it is called the last time (CBFLAG_FINAL) for a specific block. The CBFLAG_DIRECT flag indicates that the BFLAG_INDIRECT bit is not active so that the callback routine will only be called once per block. When the CBFLAG_DIRECT flag is set, the CBFLAG_FIRST and CBFLAG_FINAL flags are also set.

```
#define CBFLAG_FINAL        0x0008
#define CBFLAG_FIRST        0x0004
#define CBFLAG_DIRECT       0x0001
```

A callback routine also has a boolean return parameter in register R0. If the return value is non-zero, the subsequent memory DMA does not execute. When the CBFLAG_DIRECT flag is set, the return value has no effect.

Error Handler

While the default handler simply puts the processor into idle mode, an initcode routine can overwrite this pointer to create a customized error handler. The expected prototype is

```
void ErrorFunction (ADI_BOOT_DATA* pBootStruct, void
*pFailingAddress);
```

Use an initcode to write the entry address of the error routine to the `pErrorFunction` pointer in the `ADI_BOOT_DATA` structure. The error handler has access to the boot structure and receives the instruction address that triggered the error.

CRC Checksum Calculation

The ADSP-BF59x Blackfin processors provide an initcode and a callback routine in ROM that can be used for CRC32 checksum generation during boot time. The checksum routine only verifies the payload data of the blocks. The block headers are already protected by the native XOR checksum mechanism.

Before boot blocks can be tagged with the `BFLAG_CALLBACK` flag to enable checksum calculation on the blocks, the boot stream must contain an initcode block with no payload data and with the CRC32 polynomial in the block header `ARGUMENT` word.

The initcode registers a proper CRC32 wrapper to the `pCallBackFunction` pointer. The registration principle is similar to the XOR checksum example shown in “Programming Examples” on page 16-71.

Load Functions

All boot modes are processed by a common boot kernel algorithm. The major customization is done by a subroutine that must be registered to the

pLoadFunction pointer in the ADI_BOOT_DATA structure. Its simple prototype is as follows.

```
void LoadFunction (ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the following type:

```
typedef void ADI_BOOT_LOAD_FUNC (ADI_BOOT_DATA* ) ;
```

For a few scenarios some of the flags in the dFlags word of the ADI_BOOT_DATA structure, such as BFLAG_PERIPHERAL and BFLAG_SLAVE, slightly modify the boot kernel algorithm.

The boot ROM contains several load functions. One performs a memory DMA, others perform peripheral DMAs or load data from booting source by polling operation. The first is reused for fill operation and indirect booting as well.

In second-stage boot schemes, the user can create customized load functions or reuse the original BFROM_PDMA routine and modify the pDmaControlRegister, pControlRegister and dControlValue values in the ADI_BOOT_DATA structure. The pDmaControlRegister points to the DMAx_CONFIG or MDMA_Dx_CONFIG register. When the BFLAG_SLAVE flag is not set, the pControlRegister and dControlValue variables instruct the peripheral DMA routine to write the control value to the control register every time the DMA is started.

Load functions written by users must meet the following requirements.

- Protect against dByteCount values of zero.
- Multiple DMA work units are required if the dByteCount value is greater than 65536.
- The pSource and pDestination pointers must be properly updated.

In slave boot modes, the boot kernel uses the address of the dArgument field in the pHeader block as the destination for the required dummy DMAs when payload data is consumed from BFLAG_IGNORE blocks. If the

load function requires access to the block's ARGUMENT word, it should be read early in the function.

The most useful load functions BFROM_MDMA and BFROM_PDMA are accessible through the jump table. Others, do not have entries in the jump table. Their start address can be determined with the help of the hook routine when calling the respective BFROM_SPIBOOT or other functions. In this way, they can be re-purposed for runtime utilization.

Calling the Boot Kernel at Runtime

The boot kernel's primary purpose is to boot data to memory after power-up and reset cycles. However some of the routines used by the boot kernel might be of general value to the application. The boot ROM supports reuse of these routines as C-callable subroutines. Programs such as second-stage boot kernels, boot managers, and firmware update tools may call the function in the ROM at runtime. This could load entirely different applications or a fraction of an application, such as a code overlay or a coefficient array.

To call these boot kernel subroutines, the boot ROM provides an API at address 0xEF00 0000 in the form of a jump table.

When calling functions in the boot ROM, the user must ensure the presence of a valid stack following C language conventions. See the VisualDSP++ Compiler documentation for details.

Debugging the Boot Process

If the boot process fails, very little information can be gained by watching the chip from outside. In master boot modes, the interface signals can be observed. In slave boot modes only the HWAIT or the $\overline{\text{RTS}}$ signals tell about the progress of the boot process.

However, by using the emulator, there are many possibilities for debugging the boot process. The entire source code of the boot kernel is

provided with the VisualDSP++ installation. This includes the project executable (DXE) file. The LOAD SYMBOLS feature of the VisualDSP++ IDDE helps to navigate the program. Note that the content of the ROM might differ between silicon revisions. Hardware breakpoints and single-stepping capabilities are also available.

Table 16-6 identifies the program symbols in the boot kernel for debug.

Table 16-6. Boot Kernel Symbols for Debug

Symbol	Comment
_bootrom.assert.default	If the program counter halts at the IDLE instruction at the _bootrom.assert.default address, the boot kernel has detected an error condition and will not continue the boot process. A misformatted boot stream is the most likely cause of such an error. The RETS register points to the failing routine. When stepping a couple of instructions further, there is a way to ignore the error and to continue the boot process by clearing the >ASTAT register while the emulator steps over the subsequent IF CC JUMP 0 instruction.
_bootrom.bootmenu	If the emulator hits a hardware breakpoint at the _bootrom.bootmenu address, this indicates that a valid boot mode is being used.
_bootrom.bootkernel.entry	If the emulator hits a hardware breakpoint at the _bootrom.bootkernel.entry label, this indicates that device detection or autobaud returned properly.
_bootrom.bootkernel.breakpoint	This is a good address to place a hardware breakpoint. The boot kernel loads a new block header at this breakpoint. The block header can be watched at address 0xFF80 7FF0 or wherever the pHeader points to.
_bootrom.bootkernel.initcode	All payload data of the current block is loaded by the time the program passes the _bootrom.bootkernel.initcode label. The boot kernel is about to interrogate the BFLAG_INIT flag. If set, the initcode can be debugged.
_bootrom.bootkernel.exit	Once the boot kernel arrives at the _bootrom.bootkernel exit label, it detects a BFLAG_FINAL flag. After some housekeeping, it jumps to the EVT1 vector.

The boot kernel also generates a circular log file in scratch pad memory. While the `pLogBuffer` and the `dLogByteCount` variables describe the location and dimension of the log buffer, the `pLogCurrent` points to the next free location in the buffer. The log file is updated whenever the kernel passes the `_bootrom.bootkernel.breakpoint` label.

At each pass, nine 32-bit words are written to the log file, as follows.

- block code word (`dBlockCode`) of the block header
- target address (`pTargetAddress`) of the block header
- byte count (`dByteCount`) of the block header
- argument word (`dArgument`) of the block header
- source pointer (`pSource`) of the boot stream
- block count (`dBlockCount`)
- internal copy of the `dBlockCode` word OR'ed with `dFlags`
- content of the `SEQSTAT` register
- 0xFFFF FFFA (-6) constant

The ninth word is overwritten by the next entry set, so that 0xFFFF FFFA always marks the last entry in the log file.

Most of the data structures used by the boot kernel reside on the stack in scratchpad memory. While executing the boot kernel routine (excluding subroutines), the `P5` points to the `ADI_BOOT_DATA` structure. Type "`(ADI_BOOT_DATA*) $P5`" in the VisualDSP++ expression window to see the structure content.

Boot Management

Blackfin processor hardware platforms may be required to run different software at different times. An example might be a system with at least one application and one in-the-field firmware upgrade utility. Other systems may have multiple applications, one starting then terminating, to be replaced by another application. Conditional booting is called boot management. Some applications may self-manage their booting rules, while others may have a separate application that controls the process, namely a boot manager.

In a master boot mode where the on-chip boot kernel loads the boot stream from memory, the boot manager is a piece of Blackfin software which decides at runtime what application is booted next. This may simply be based on the state of a GPIO input pin interrogated by the boot manager, or it may be the conclusion of complex system behavior.

Slave boot scenarios are different from master boot scenarios. In slave boot modes, the host masters boot management by setting the Blackfin processor to reset and then applying alternate boot data. Optionally, the host could alter the `BMODE` configuration pins, resulting in little impact to the Blackfin processor since the intelligence is provided by the host device.

Booting a Different Application

The boot ROM provides a set of user-callable functions that help to boot a new application (or a fraction of an application). Usually there is no need for the boot manager to deal with the format details of the boot stream.

One such function is `BFROM_SPIBOOT`, which is discussed in "SPI Master Boot Modes" on page 16-40.

The user application, the boot manager application, or an initcode can call these functions to load the requested boot data. Using the `BFLAG_RETURN`

flag the user can control whether the routine simply returns to the calling function or executes the loaded application immediately.

These ROM functions expect the start address of the requested boot stream as an argument. For BFROM_SPIBOOT, it is a serial address. The SPI function can also accept the code for the GPIO pin that controls the device select strobe of the SPI memory.

Multi-DXE Boot Streams

If the start addresses of all the boot streams are predefined, the boot manager needs only to call the ROM functions directly. However since the addresses tend to vary from build to build they may have to be calculated at runtime.

In the world of the VisualDSP++ elfloader, a boot stream is always generated from a DXE file. It is therefore common to talk about multi-DXE or multi-application booting. When the elfloader utility accepts multiple DXE files on its command line, it generates a contiguous boot image by default. The second boot stream is appended immediately to the first one. Since the utility updates the ARGUMENT field of all BFLAG_FIRST blocks, the ARGUMENT field of a BFLAG_FIRST block is called next-DXE pointer (NDP).

The next-DXE pointer of the first DXE boot stream points relatively to the start address of the second DXE boot stream. A multi-DXE boot image can be seen as a linked list of boot streams. The next-DXE pointer of the last DXE boot stream points relatively to the next free address.

Figure 16-7 on page 16-37 shows a screenshot of the Blackfin loader file viewer utility. The LdrViewer utility is not part of the VisualDSP++ tools suite. It is a third-party freeware product available on http://www.dolomitics.com.

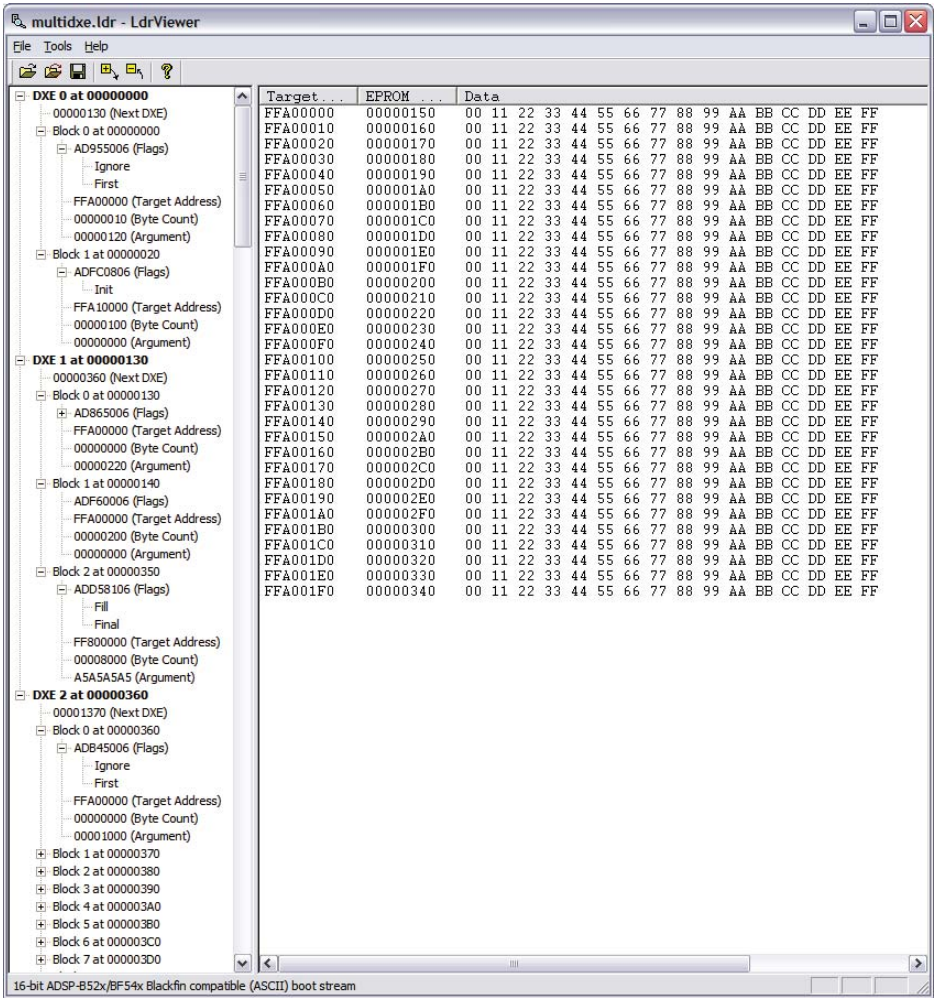


Figure 16-7. LdrViewer Screen Shot

Determining Boot Stream Start Addresses

The ROM functions (`BFROM_SPIBOOT` and others) not only allow the application to boot a subroutine residing at a given start address, they also assist in walking through linked multi-DXE streams.

When the BFLAG_NEXTDXE bit in dFlags is set and these functions are called, the system does not boot but instead walks though the boot stream following the next-DXE pointers. The dBlockCount parameter can be used to specify the DXE of interest. The routines then return the start address of the requested DXE's boot stream.

Initialization Hook Routine

When the ROM functions (BFROM_SPIBOOT and others) are called, they create an instance of the ADI_BOOT_DATA structure on the stack and fill the items with default values. If the BFLAG_HOOK is set, the boot kernel invokes a callback routine which was passed as the fourth argument of the ROM routines, after the default values have been filled. The hook routine can be used to overwrite the default values. Every hook routine should fit the prototype:

```
void hook (ADI_BOOT_DATA* pBS);
```

The VisualDSP++ header files define the ADI_BOOT_HOOK_FUNC type the following way:

```
typedef void ADI_BOOT_HOOK_FUNC (ADI_BOOT_DATA*);
```

The hook function also gives access to the DMA load function used by the respective boot mode, which can be used for general purposes at runtime. For example, in the BFROM_SPIBOOT case, an instance of the load function:

```
ADI_BOOT_LOAD_FUNC *pSpiLoadFunction;
```

can be initialized by equipping the hook function with the instruction:

```
pSpiLoadFunction = pBS->pLoadFunction;
```

Specific Boot Modes

This section discusses individual boot modes and the required hardware connections.

The boot modes differ in terms of the booting source— for example whether data is loaded through the SPI or the parallel interface. Boot modes can also be grouped into slave boot modes and master boot modes.

In slave boot modes, the Blackfin processor functions as a slave to any host device, which is typically another embedded processor, an FPGA device or even a desktop computer. Likely, the Blackfin processor $\overline{\text{RESET}}$ input is controlled by the host device. So, usually the host sets $\overline{\text{RESET}}$ first, then waits until the host senses the HWAIT output, and finally provides the boot data.

If a Blackfin processor, configured to operate in any of the slave boot modes, awakens from hibernate, it cannot boot by its own control. A feedback mechanism has to be implemented at the system level to inform the host device whether the processor is in hibernate state or not. The HWAIT strobe is an important primitive in such systems.

In the master boot modes, the Blackfin processor usually does not need to be synchronized and can load the boot data by itself. Master modes typically read from memory. This can be serial memory that is read through SPI interfaces.

Whether from the host (slave booting mode) or from memory (master booting mode), the boot source does not need to know about the structure of the boot stream.

No Boot Mode

When the BMODE pins are all tied low (BMODE = 000), the Blackfin processor does not boot. Instead, it executes an IDLE instruction, preventing it from executing any instructions provided by the regular boot source. The purpose of this mode is to bring the processor up to a clean state after reset.

When connecting an emulator and starting a debug session, the processor awakens from an idle due to the emulation interrupt and can be debugged in the normal manner.

The no boot mode is not the same as the bypass mode featured by the ADSP-BF53x Blackfin processor.

SPI Master Boot Modes

The ADSP-BF59x processors feature booting from off-chip SPI memory using either SPI1 or SPI0.

The external SPI boot modes (BMODE = 010 or 100) boot from SPI memories connected to the SPI1_SSEL5 (SPI1) or SPI0_SSEL2 (SPI0) interface. 8-, 16-, 24-, and 32-bit address words are supported. Standard SPI memories are read using either the standard 0x03 SPI read command or the 0x0B SPI fast read command.

Unlike other Blackfin processors, the ADSP-BF59x Blackfin processors have no special support for DataFlash devices from Atmel. Nevertheless, DataFlash devices can be used for booting and are sold as standard 24-bit addressable SPI memories. They also support the fast read mode. If used for booting, DataFlash memory must be programmed in the power-of-2 page mode.

For booting, the SPI memory is connected as shown in Figure 16-8 (SPI1) or Figure 16-9 (SPI0).

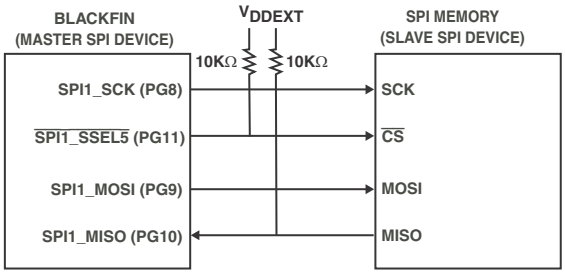


Figure 16-8. Blackfin to SPI1 Memory Connections

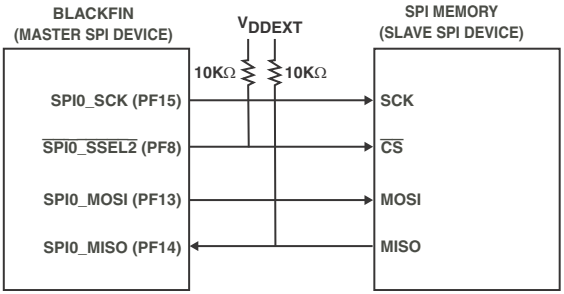


Figure 16-9. Blackfin to SPI0 Memory Connections

The pull-up resistor on the MISO line is required for automatic device detection. The pull-up resistor on the $\overline{\text{SPI1\_SSEL5}}$ or $\overline{\text{SPI0\_SSEL2}}$ line ensures that the memory is in a known state when the Blackfin GPIO is in a high-impedance state (for example, during reset). A pull-down resistor on the SPI clock line (SPI1_SCK or SPI0_SCK) displays cleaner oscilloscope plots during debugging.

For SPI master boot, the SPE, MSTR and SZ bits are set in the SPI1_CTL or SPI0_CTL register. For details see Chapter 13, "SPI-Compatible Port Controller". With TIMOD = 2, the receive DMA mode is selected. Clearing both the CPOL and CPHA bits results in SPI mode 0. The boot kernel does not allow SPI1 or SPI0 hardware to control the $\overline{\text{SPI1\_SSEL5}}$ or $\overline{\text{SPI0\_SSEL2}}$ pin. Instead, this pin is toggled in GPIO mode by software. Initialization code is allowed to manipulate the uwSsel variable in the ADI_BOOT_DATA structure to extend the boot mechanism to a second SPI memory connected to another GPIO pin.

By default, the boot kernel sets the SPI1_BAUD or SPI0_BAUD register to a value of 133, resulting in a bit rate of SCLK/266 (as shown in Table 16-7).

Table 16-7. Bit Rate

SPI_BAUD	Bit Rate
133	SCLK/(2x133) << default
Reserved	
2	SCLK/(2x2)
4	SCLK/(2x4)
8	SCLK/(2x8)
16	SCLK/(2x16)
32	SCLK/(2x32)
64	SCLK/(2x64)

Similarly, the boot kernel uses the standard 0x03 SPI read command, by default.

SPI Device Detection Routine

Because BMODE = 010 or 100 supports booting from various SPI memories, the boot kernel automatically detects what type of memory is connected. To determine whether the SPI memory device requires an 8-, 16-, 24- or 32-bit addressing scheme, the boot kernel performs a device detection sequence prior to booting. The MISO signal requires a pull-up resistor, since the routine relies on the fact that memories do not drive their data outputs unless the right number of address bytes are received.

Initially, the boot kernel transmits a read command (either 0x03 or 0x0B) on the MOSI line, which is immediately followed by two zero bytes. Once the transmission is finished, the boot kernel interrogates the data received on the MISO line. If it does not equal 0xFF (usually a DMACODE value of 0x01 is expected), then an 8-bit addressable device is assumed.

If the received value equals 0xFF, it is assumed that the memory device has not driven its data output yet and that the 0xFF value is due to the pull-up resistor. Thus, another zero byte is transmitted and the received data is

tested again. If it differs from 0xFF, either a 16-bit addressable device (standard mode) or an 8-bit addressable device (fast read mode) is assumed.

If the value still equals 0xFF, device detection continues. Device detection aborts immediately if a byte different than 0xFF is received. The boot kernel continues with normal boot operation and it re-issues a read command to read from address 0 again. The first block header is loaded by two read sequences, further block headers and block payload fields are loaded by separate read sequences.

Figure 16-10 illustrates how individual devices would behave.

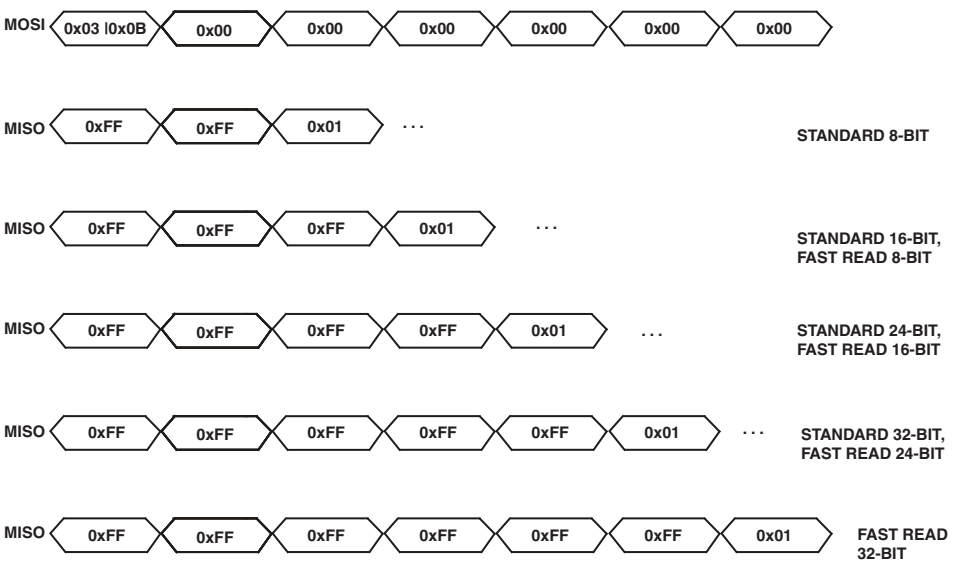


Figure 16-10. SPI Device Detection Principle

Figure 16-11 on page 16-44 shows the initial signaling when a 24-bit addressable SPI memory is connected in SPI master boot mode. After $\overline{\text{RESET}}$ releases, a 0x03 command is transmitted to the MOSI output, followed by a number of 0x00 bytes. The 24-bit addressable memory device

returns a first data byte at the fourth zero byte. Then, the device detection has completed and the boot kernel re-issues a 0x00 address to load the boot stream.

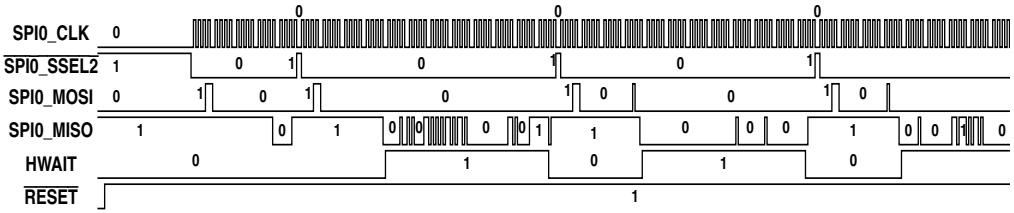


Figure 16-11. Typical SPI Master Boot Waveforms

SPI Slave Boot Mode

For SPI slave mode boot (BMODE = 011), the Blackfin processor is consuming boot data from an external SPI host device. SPI1 is configured as an SPI slave device. The hardware configuration is shown in Figure 16-12. As in all slave boot modes, the host device controls the Blackfin processor RESET input.

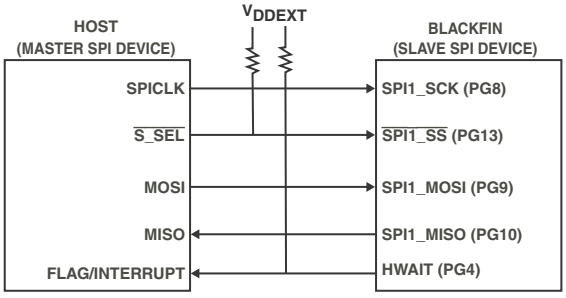


Figure 16-12. Connections Between Host (SPI Master) and Blackfin Processor (SPI Slave)

The host drives the SPI clock and is responsible for the timing. The host must provide an active-low chip select signal that connects to the SPI1_SS

input of the Blackfin processor. It can toggle with each byte transferred or remain low during the entire procedure. 8-bit data is expected. The 16-bit mode is not supported.

In SPI slave boot mode, the boot kernel sets the CPHA bit and clears the CPOL bit in the SPI1_CTL register. Therefore the MOSI pin is latched on the falling edge of the SCK (SPI clock). For details see Chapter 13, "SPI-Compatible Port Controller".

In SPI slave boot mode, HWAIT functionality is critical. When high, the resistor shown in Figure 16-12 programs HWAIT to hold off the host. HWAIT holds the host off while the Blackfin processor is in reset. Once HWAIT turns inactive, the host can send boot data. The SPI module does not provide very large receive FIFOs, so the host must test the HWAIT signal for every byte. Figure 16-14 on page 16-46 illustrates the required program flow on the host side.

Figure 16-13 shows the initial waveform for an SPI slave boot case. As soon as the Blackfin processor releases HWAIT after reset, the host device pulls the $\overline{\text{SPI1\_SS}}$ pin low and starts transmitting data. After the eighth data word has been received, the boot kernel asserts HWAIT again as it has to process the DMACODE field of the first block header. When the host detects the asserted HWAIT it gracefully finishes the transmission of the on-going word. Then, it pauses transmission until HWAIT releases again.

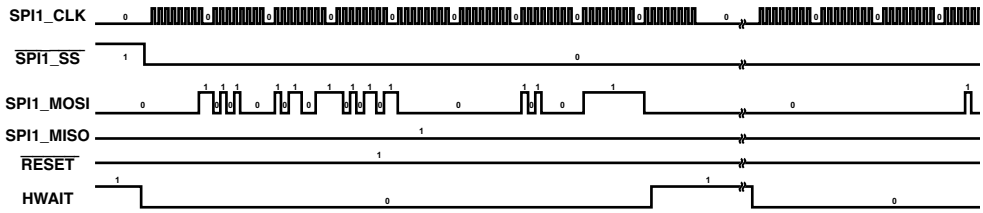


Figure 16-13. Typical SPI Slave Boot Waveforms

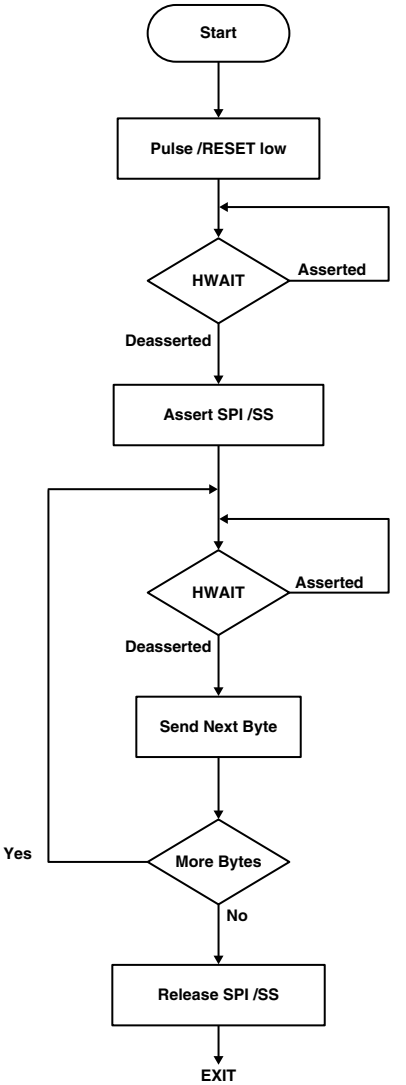


Figure 16-14. SPI Program Flow on Host Device

PPI Boot Mode

The ADSP-BF59x processors feature a 16-bit PPI boot mode (BMODE = 101). The PPI is a half-duplex bi-directional port consisting of up to 16 data lines, 3 frame synchronization signals and a clock signal.

In PPI boot mode, the PPI mode of operation is configured as follows:

- Receive mode with 1 external frame sync
- 16-bit bus width
- Data sampled on falling edge of clock
- Frame sync configured for falling edge asserted
- PPI_DELAY value of 0x0

The external frame sync signal is on PPI_FS1. This signal is driven low by the host at the start of a data transfer with a 16-bit word being transferred on each PPI_CLK cycle that the PPI_FS1 signal is asserted low.

In order to simplify the PPI host design, PPI boot mode also configures Timer1 for PWM mode of operation. The PWM circuits of the timer are configured to be clocked by the externally provided PPI_CLK signal allowing for arbitrary pulse widths and pulse periods to be programmed thus simulating an internally generated frame sync signal on the PPI_FS2 signal. This configuration lets the processor inform the host when the processor is ready to receive data and also how much data is expected. This feature removes the need for the host to process the actual contents of the boot stream to identify the size of the data transfer.

The PPI host can synchronize the PPI_FS2 signal to PPI_CLK signal and initiate all data transfers accordingly. The PPI_FS2 signal can be looped back to the PPI_FS1. (See Figure 16-15.)

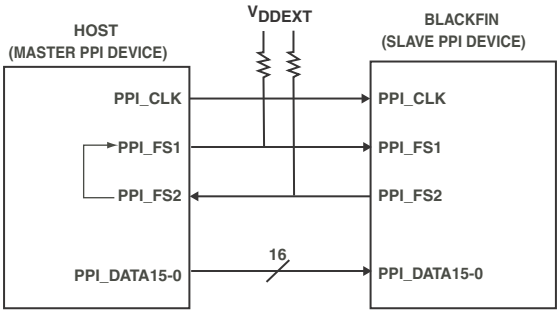


Figure 16-15. PPI Slave Boot Mode Connections

The Timer1 is configured to generate a periodic pulse as opposed to a single shot pulse. The pulse period is set to the maximum of 0xFFFFFFFF allowing for any transfer size supported by the kernel. Note the current 16-bit DMA X Count limits the maximum width of a pulse to 0xFFFF words.

After completion of the DMA transfer, the PWM_OUT out mode is terminated and cleared in the required manner. This mode of operation does impose some restrictions on the amount of time that the PPI host device can hold off a transfer. If a DMA transfer consists of 0xFFFF words, the timer period will be reached 0xFFFF0000 PPI_CLK cycles after the de-assertion of the PPI_FS2/TMR1 signal. This will result in the generation of an identical PPI_FS2/TMR1 pulse if the DMA transfer has not completed and the PWM_OUT timer has not been disabled.

In the unlikely event that a user requires a transfer to be held off for this significant amount of time, the PPI host must be able to ignore any further PPI_FS2/TMR1 assertions until the currently pending transaction that was delayed has completed. If the master is not capable of ignoring further PPI_FS2/TMR1 assertions, the master must ensure that the DMA completes allowing for the PWM_OUT timer to be disabled prior to the completion of the timer pulse period of 0xFFFFFFFF PPI_CLK cycles.

After PPI boot completion the PPI interface is disabled and the PPI_CONTROL register is cleared, this register-clearing operation is not done for the Timer1 registers. Although the timer is disabled, the TIMER1_CONFIG register is not reloaded with the default reset value.

UART Slave Mode Boot

Figure 16-16 on page 16-49 shows the interconnection required for booting. The figure does not show physical line drivers and level shifters that are typically required to meet the individual UART-compatible standards.

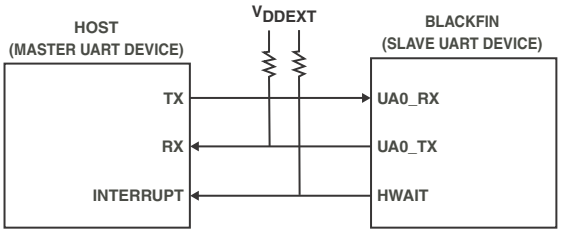


Figure 16-16. UART Slave Boot Mode Connections

For BMODE = 101, the ADSP-BF59x processor consumes boot data from a UART host device connected to the UART0 interface.

The host downloads programs formatted as boot streams using an auto-baud detection sequence. The host selects a bit rate within the UART clocking capabilities. To determine the bit rate when performing the auto-baud, the boot kernel expects an "@" character (0x40, eight data bits, one start bit, one stop bit, no parity bit) on the UART UA0_RX input. The boot kernel acknowledges, and the host then downloads the boot stream. The acknowledgement consists of four bytes: 0xBF, UARTx_DLL, UARTx_DLH, 0x00. The host is requested to not send further bytes until it has received the complete acknowledge string. Once the 0x00 byte is received, the host can send the entire boot stream. The host should know the total byte count of the boot stream, but it is not required to have any knowledge

about the content of the boot stream. Further information regarding auto-baud detection is given in “Autobaud Detection” on page 11-13.

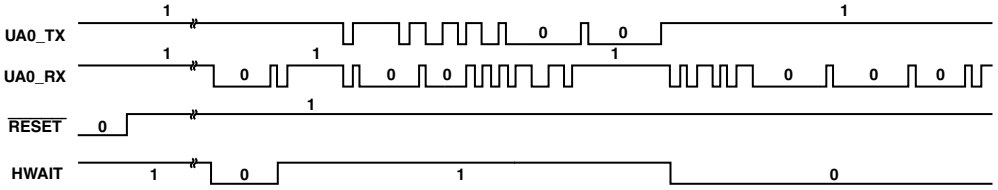


Figure 16-17. UART Autobaud Waveform

When the boot kernel is processing fill or initcode blocks it might require extra processing time and needs to hold the host off from sending more data. This is signalled with the HWAIT output. When equipped with a pull-up resistor the HWAIT signal imitates the behavior of an $\overline{\text{UA0\_RTS}}$ output and could be connected to the $\overline{\text{CTS}}$ input of the booting host.The host is not allowed to send data until HWAIT turns inactive after a reset cycle. Therefore a pulling resistor on the HWAIT signal is required.

If the resistor pulls to ground, the host must pause transmission when HWAIT is low and is permitted to send when HWAIT is high. A pull-up resistor inverts the signal polarity of HWAIT. The host should test HWAIT at every transmitted byte.

Figure 16-18 shows the initial case of the UART boot mode. As soon as HWAIT releases after reset, the boot kernel expects to receive a 0x40 byte for bit rate detection. After the bit rate is known, the UART is enabled and the kernel transmits for bytes.

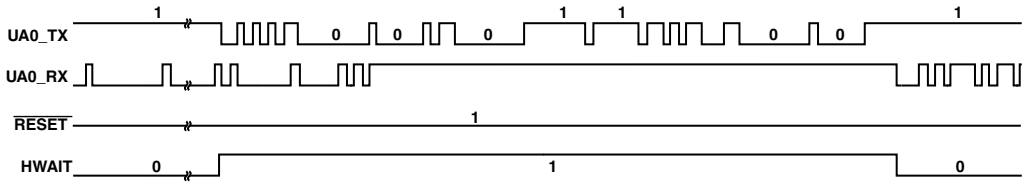


Figure 16-18. UART Boot - Host relying on HWAIT

For UART boot, it is not obvious on how to change the PLL by an initcode routine. This is because the UARTx_DLL and UARTx_DLH registers have to be updated to keep the required bit rate constant after the SCLK frequency has changed. It must be ensured that the host does not send data while the PLL is changing. The initcode examples provided along with the VisualDSP++ tools installation demonstrate how this can be accomplished.

L1 ROM Boot Mode

This boot mode may be chosen only by customers who have the custom product with its L1 IROM mask programmed by the factory.

In this boot mode, the processor starts instruction execution at address 0xFFA1 0000 of the on-chip L1 instruction ROM, entirely bypassing the boot ROM. This option provides users with full control over the booting process. It is highly recommended that users review the contents of the factory provided boot ROM in the course of developing their own L1 ROM boot sequence.

Analog Devices does not provide technical support for custom boot code development. For more information about custom product and custom IROM mask production, contact your Analog Devices representative.

Reset and Booting Registers

Two registers are used for reset and booting—the software reset register (SWRST) and the system reset configuration register (SYSCR).

Software Reset (SWRST) Register

A software reset can be initiated by setting bits [2:0] in the system software reset field in the software reset register (SWRST) shown in Figure 16-19 on page 16-52.

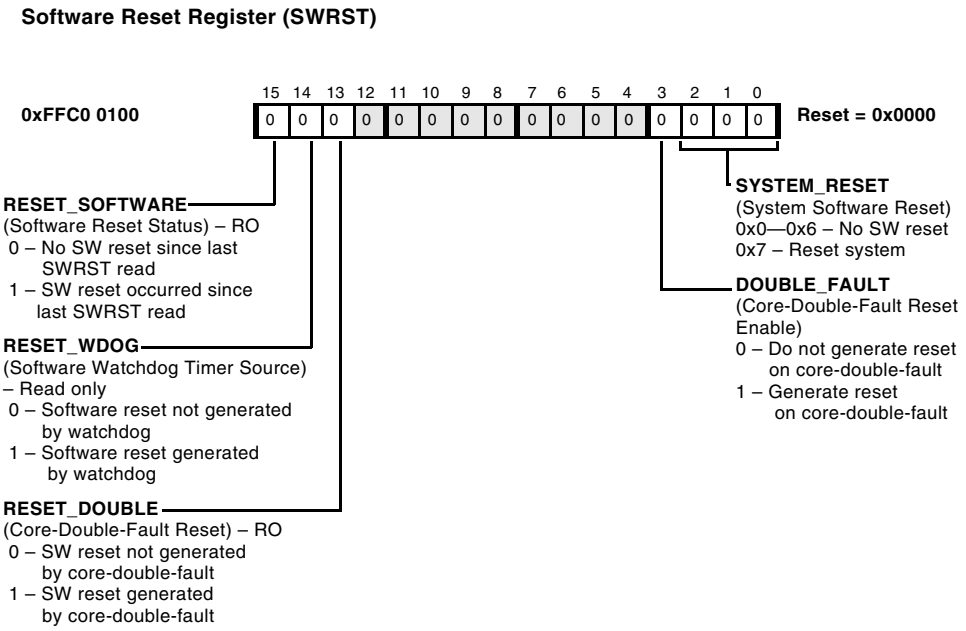

Software Reset Register (SWRST)
0xFFC0 0100
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000
RESET_SOFTWARE
(Software Reset Status) – RO
0 – No SW reset since last SWRST read
1 – SW reset occurred since last SWRST read
RESET_WDOG
(Software Watchdog Timer Source) – Read only
0 – Software reset not generated by watchdog
1 – Software reset generated by watchdog
RESET_DOUBLE
(Core-Double-Fault Reset) – RO
0 – SW reset not generated by core-double-fault
1 – SW reset generated by core-double-fault
SYSTEM_RESET
(System Software Reset)
0x0—0x6 – No SW reset
0x7 – Reset system
DOUBLE_FAULT
(Core-Double-Fault Reset Enable)
0 – Do not generate reset on core-double-fault
1 – Generate reset on core-double-fault

Figure 16-19. Software Reset Register

Bit 3 can be used to generate a reset upon core-double-fault. A core-double-fault resets both the core and the peripherals, but not most of the DPMC. Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 indicates the software watchdog timer has generated the software reset. Bit 13 indicates the core-double-fault has generated the software reset. Bits [15:13] are read-only and cleared when the register is read. Reading the SWRST also clears bits [15:13] in the SYSCR register. Bits [3:0] are read/write.

Only writing to bits[2:0], resets only the modules in the SCLK domain. It does not clear the core. The program executes normally at the instruction after the MMR write to SWRST. The system is kept in the reset state as long as the bits[2:0] are set to b#111. To release reset, write a zero again. Examples for this are available in assembly (Listing 16-1 on page 16-71) and C (Listing 16-2 on page 16-72). It is not recommended that this functionality be used directly. Rather, call the ROM function bfrom_SysControl() to perform a system reset.

System Reset Configuration (SYSCR) Register

The software reset configuration register (SYSCR) is shown in Figure 16-20 on page 16-54.

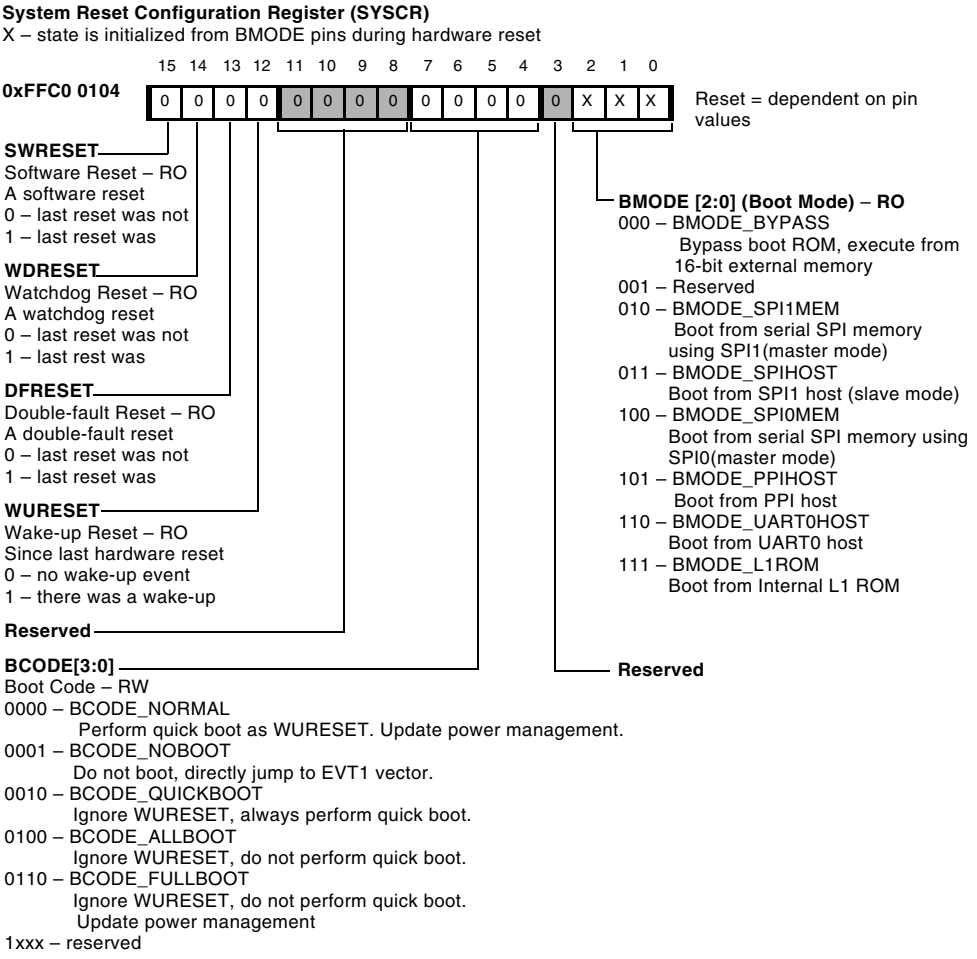

System Reset Configuration Register (SYSCR)
X – state is initialized from BMODE pins during hardware reset
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xFFC0 0104
0 0 0 0 0 0 0 0 0 0 0 0 0 X X X
Reset = dependent on pin values
SWRESET
Software Reset – RO
A software reset
0 – last reset was not
1 – last reset was
WDRESET
Watchdog Reset – RO
A watchdog reset
0 – last reset was not
1 – last rest was
DFRESET
Double-fault Reset – RO
A double-fault reset
0 – last reset was not
1 – last reset was
WURESET
Wake-up Reset – RO
Since last hardware reset
0 – no wake-up event
1 – there was a wake-up
Reserved
BCODE[3:0]
Boot Code – RW
0000 – BCODE_NORMAL
Perform quick boot as WURESET. Update power management.
0001 – BCODE_NOBOOT
Do not boot, directly jump to EVT1 vector.
0010 – BCODE_QUICKBOOT
Ignore WURESET, always perform quick boot.
0100 – BCODE_ALLBOOT
Ignore WURESET, do not perform quick boot.
0110 – BCODE_FULLBOOT
Ignore WURESET, do not perform quick boot.
Update power management
1xxx – reserved
BMODE [2:0] (Boot Mode) – RO
000 – BMODE_BYPASS
Bypass boot ROM, execute from 16-bit external memory
001 – Reserved
010 – BMODE_SPI1MEM
Boot from serial SPI memory using SPI1(master mode)
011 – BMODE_SPIHOST
Boot from SPI1 host (slave mode)
100 – BMODE_SPI0MEM
Boot from serial SPI memory using SPI0(master mode)
101 – BMODE_PPIHOST
Boot from PPI host
110 – BMODE_UART0HOST
Boot from UART0 host
111 – BMODE_L1ROM
Boot from Internal L1 ROM
Reserved

Figure 16-20. System Reset Configuration Register

The values sensed from the BMODE[2:0] pins are mirrored into the system reset configuration register (SYSCR). The values are available for software access and modification after the hardware reset sequence. Software can modify only bits[7:4] in this register to customize boot processing upon a software reset.

The WURESET indicates whether there was a wake up from hibernate since the last hardware reset. The bit cannot be cleared by software.

The bits [15:13] are exact copies of the same bits in the SWRST register. Unlike the SWRST register, SYSCR can be read without clearing these bits. Reading SWRST also causes SYSCR[15:13] to clear.

Boot Code Revision Control (BK_REVISION)

The boot ROM reserves the 32-bits at address 0xEF00 0040 for a four byte version code as shown in Figure 16-21.

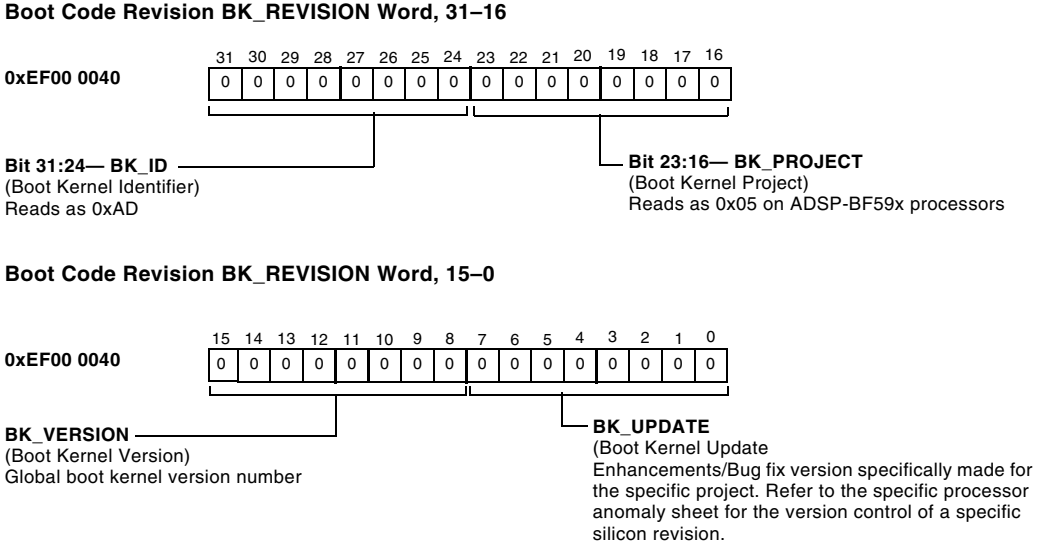

Boot Code Revision BK_REVISION Word, 31–16
0xEF00 0040
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Bit 31:24— BK_ID
(Boot Kernel Identifier)
Reads as 0xAD
Bit 23:16— BK_PROJECT
(Boot Kernel Project)
Reads as 0x05 on ADSP-BF59x processors
Boot Code Revision BK_REVISION Word, 15–0
0xEF00 0040
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BK_VERSION
(Boot Kernel Version)
Global boot kernel version number
BK_UPDATE
(Boot Kernel Update
Enhancements/Bug fix version specifically made for the specific project. Refer to the specific processor anomaly sheet for the version control of a specific silicon revision.

Figure 16-21. Boot Code Revision Code (BK_REVISION)

Boot Code Date Code (BK_DATECODE)

The boot ROM reserves the 32-bits at address 0xEF00 0050 for the build date as shown in Figure 16-22.

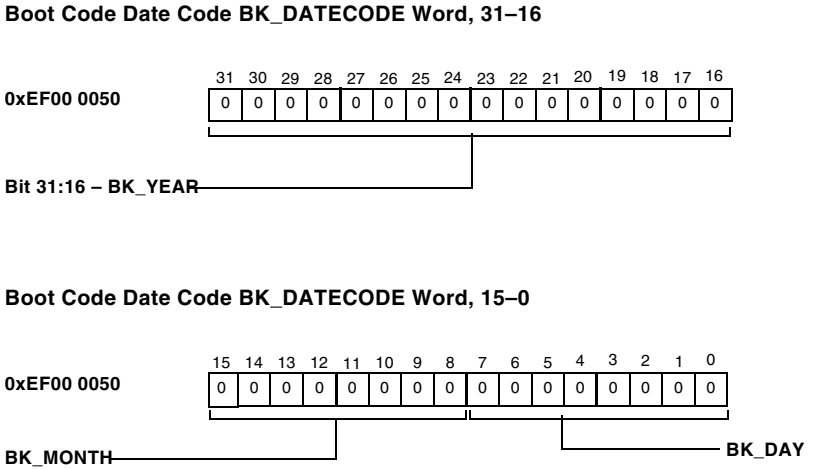


Figure 16-22. Boot Code Date Code (BK_DATECODE)

Zero Word (BK_ZEROS)

The boot ROM reserves the 32-bits at address 0xEF00 0048 which always reads as 0x0000 000 as shown in Figure 16-23.

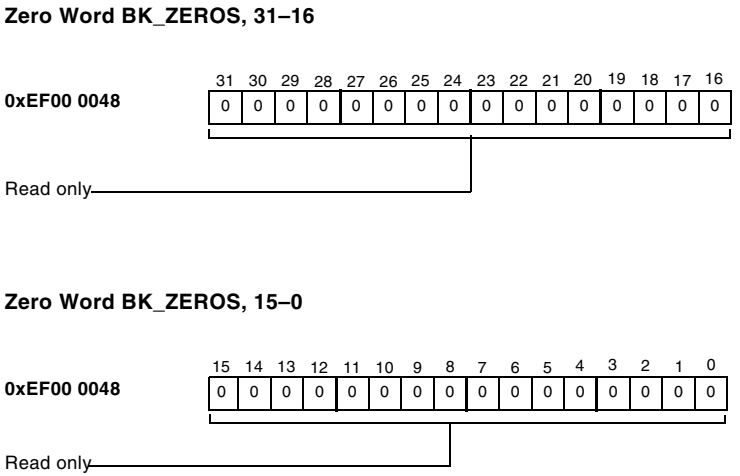


Figure 16-23. Zero Word (BK_ZEROS)

Ones Word (BK_ONES)

The boot ROM reserves the 32-bits at address 0xEF00 004C which always reads 0xFFFF FFFF as shown in Figure 16-24.

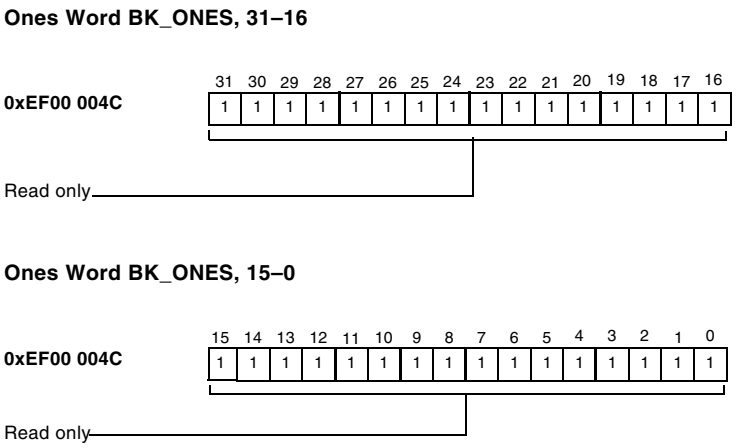


Figure 16-24. Ones Word (BK_ONES)

Data Structures

The boot kernel uses specific data structures for internal processing. Advanced users can customize the booting process by changing the content of the structure within the initcode routines. This section uses C language definitions for documentation purposes. VisualDSP++ users can use these structures directly in assembly programs by using the `.IMPORT` directive. The structures are supplied by the `bfrom.h` header file in your VisualDSP++ installation directory.

ADI_BOOT_HEADER

```
typedef struct {
    s32  dBlockCode;
    void* pTargetAddress;
    s32  dByteCount;
    s32  dArgument;
} ADI_BOOT_HEADER;
```

The structure `ADI_BOOT_HEADER` is used by the boot kernel to load and process a block header.

Every block header is loaded to L1 data memory location 0xFF80 7FF0–0xFF80 7FFF first or where `pHeader` points to. There it is analyzed by the boot kernel.

ADI_BOOT_BUFFER

```
typedef struct {
    void*  pSource;
    s32    dByteCount;
} ADI_BOOT_BUFFER;
```

The structure `ADI_BOOT_BUFFER` is used for any kind of buffer. For the user, this structure is important when implementing advanced callback mechanisms.

ADI_BOOT_DATA

```
typedef struct {
    void*  pSource;
    void*  pDestination;
    s16*   pControlRegister;
    s16*   pDmaControlRegister;
    s32    dControlValue;
```

```
    s32    dByteCount;
    s32    dFlags;
    s16    uwDataWidth;
    s16    uwSrcModifyMult;
    s16    uwDstModifyMult;
    s16    uwHwait;
    s16    uwSsel;
    s16    uwUserShort;
    s32    dUserLong;
    s32    dReserved;

    ADI_BOOT_ERROR_FUNC*  pErrorFunction;
    ADI_BOOT_LOAD_FUNC*  pLoadFunction;
    ADI_BOOT_CALLBACK_FUNC*  pCallBackFunction;
    ADI_BOOT_HEADER* pHeader;
    void*  pTempBuffer;
    void*  pTempCurrent;
    s32    dTempByteCount;
    s32    dBlockCount;
    s32    dClock;
    void*  pLogBuffer;
    void*  pLogCurrent;
    s32    dLogByteCount;
} ADI_BOOT_DATA;
```

The structure `ADI_BOOT_DATA` is the main data structure. A pointer to a `ADI_BOOT_DATA` structure is passed to most complex subroutines, including load functions, initcode, and callback routines. The structure has two parts. While the first is closely related to internal memory load routines, the second provides access to global boot settings.

Table 16-8 on page 16-61 describes the data structures.

Table 16-8. Structure Variables, ADI_BOOT_DATA

Variable	Description
pSource	In the context of the boot kernel, the pSource pointer points either to the start address of the entire boot stream or to the header of the next boot block. In the context of memory load routines pSource points to the source address of the DMA work unit.
pDestination	The pDestination pointer is only used in memory load routines. It points to the destination address of the DMA work unit. It points to either 0xFF80 7FF0 when a header is loaded, or the target address when the payload data is loaded.
pControlRegister	This pointer holds the MMR address of the peripheral's main control register (for example UARTx_LCR or SPIx_CTL)
pDmaControlRegister	This pointer holds the MMR address of the DMAx_CONFIG register for the DMA channel in use.
dControlValue	The lower 16 bits of this value are written to the pControlRegister location each time a DMA work unit is started.
dByteCount	Number of bytes to be transferred.
dFlags	The lower 16 bits of this variable hold the lower 16 bits of the current block code. The upper 16 bits hold global flags. See "dFlags Word" on page 16-63.
uwDataWidth	This instructs the memory load routine to use: 0 – 8-bit DMA 1 – 16-bit DMA 2 – 32-bit DMA
uwSrcModifyMult	This is the multiplication factor used by the DMA source. A value of 1 sets the source modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwDstModifyMult	This is the multiplication factor used by the DMA destination. A value of 1 sets the destination modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwHwait	This 16-bit value holds the GPIO used for HWAIT signaling. The PG4 pin is configured as HWAIT signal on ADSP-BF59x processors. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.

Table 16-8. Structure Variables, ADI_BOOT_DATA (Continued)

Variable	Description
uwSsel	This 16-bit value holds the GPIO used for SPI slave select. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.
uwUserShort	The programmer can use this 16-bit value for passing parameters between modules of a customized booting scheme.
dUserLong	The programmer can use this 32-bit value for passing parameters between modules of a customized booting scheme.
dReserved	This 32-bit value is reserved for future development.
pErrorFunction	This is the pointer to the error handler. See "Error Handler" on page 16-30.
pLoadFunction	This is the pointer to the function responsible for loading data. See "Load Functions" on page 16-30
pCallBackFunction;	This is the pointer to the callback function. See "Callback Routines" on page 16-27
pHeader	The pHeader pointer holds the address for intermediate storage of the block header. By default this value is set to 0xFF80 7FF0.
pTempBuffer	This pointer tells the boot kernel what memory to use for intermediate storage when the BFLAG_INDIRECT flag is set for a given block. The pointer defaults to 0xFF80 7F00. The value can be modified by the initcode routine, but there would be some impact to the VisualDSP++ tools.
pTempCurrent	Defaults to the pTempBuffer value. A load function can modify this value to manipulate subsequent callback and memory DMA routines.
dTempByteCount	This is the size of the intermediate storage buffer used when the BFLAG_INDIRECT flag is set for a given block. This value defaults to 256 and can be modified by an initcode routine. When increasing this value, the pTempBuffer must also be changed since by default the block is at the end of a physical data memory block.
dBlockCount	This 32-bit variable counts the boot blocks that are processed by the boot kernel. If the user sets this value to a negative value, the boot kernel exits when the variable increments to zero.
dClock	The dClock variable holds information about the clock divider used by individual (serial) boot modes.

Table 16-8. Structure Variables, `ADI_BOOT_DATA` (Continued)

Variable	Description
`pLogBuffer`	Pointer to the circular log buffer. By default the log buffer resides in L1 scratch pad memory at address 0xFFB0 0400.
`pLogCurrent`	Pointer to the next free entry of the circular log buffer.
`dLogByteCount`	Size of the circular log buffer, default is 0x400 bytes.

dFlags Word

Figure 16-26 and Figure 16-25 describe the `dFlags` word. `dFlags [15-0]` is a copy of Block Code[15–0] of the block currently being processed.

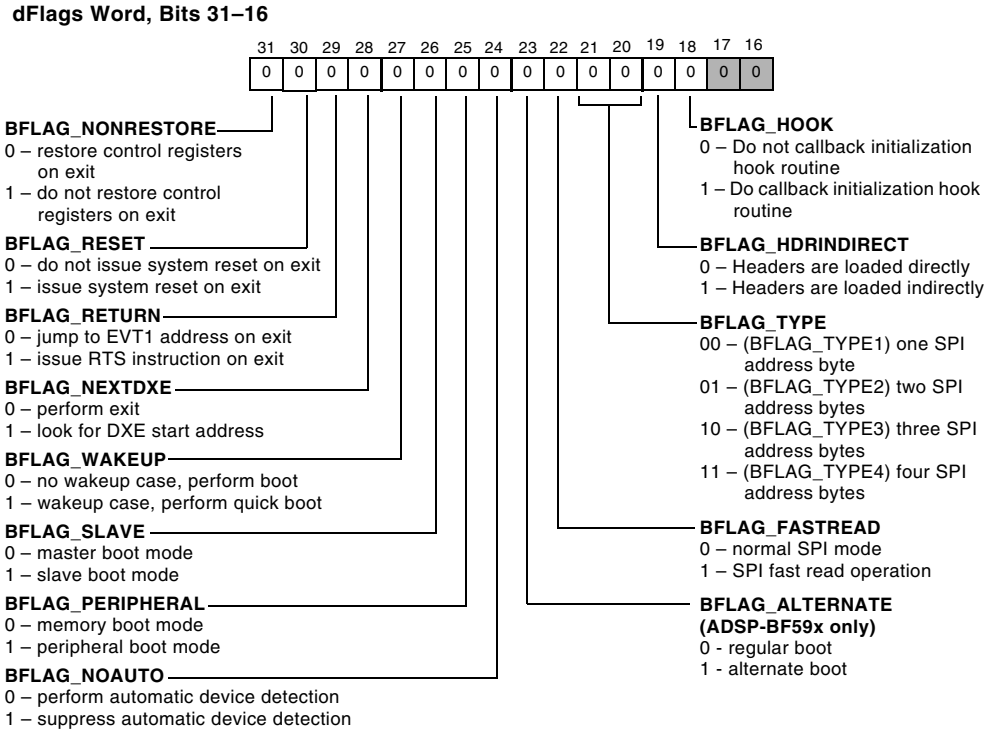

dFlags Word, Bits 31–16
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BFLAG_NONRESTORE
0 – restore control registers on exit
1 – do not restore control registers on exit
BFLAG_RESET
0 – do not issue system reset on exit
1 – issue system reset on exit
BFLAG_RETURN
0 – jump to EVT1 address on exit
1 – issue RTS instruction on exit
BFLAG_NEXTDXE
0 – perform exit
1 – look for DXE start address
BFLAG_WAKEUP
0 – no wakeup case, perform boot
1 – wakeup case, perform quick boot
BFLAG_SLAVE
0 – master boot mode
1 – slave boot mode
BFLAG_PERIPHERAL
0 – memory boot mode
1 – peripheral boot mode
BFLAG_NOAUTO
0 – perform automatic device detection
1 – suppress automatic device detection
BFLAG_HOOK
0 – Do not callback initialization hook routine
1 – Do callback initialization hook routine
BFLAG_HDRINDIRECT
0 – Headers are loaded directly
1 – Headers are loaded indirectly
BFLAG_TYPE
00 – (BFLAG_TYPE1) one SPI address byte
01 – (BFLAG_TYPE2) two SPI address bytes
10 – (BFLAG_TYPE3) three SPI address bytes
11 – (BFLAG_TYPE4) four SPI address bytes
BFLAG_FASTREAD
0 – normal SPI mode
1 – SPI fast read operation
BFLAG_ALTERNATE
(ADSP-BF59x only)
0 - regular boot
1 - alternate boot

Figure 16-25. dFlags Word (Bits 31–16)

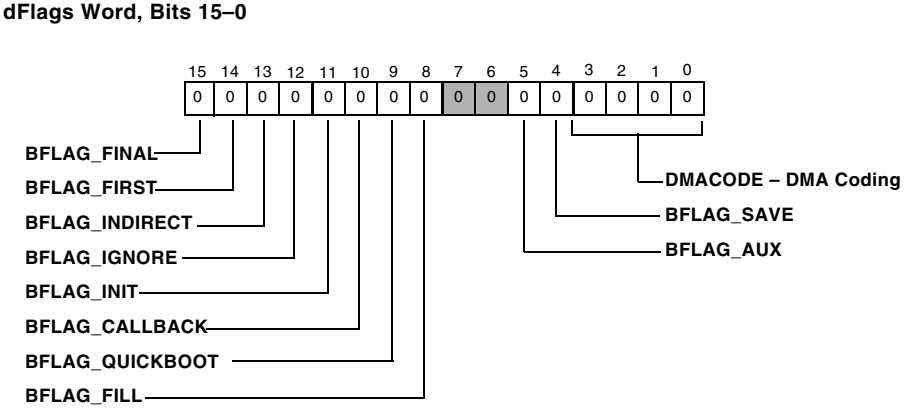


Figure 16-26. dFlags Word (Bits 15–0)

Callable ROM Functions for Booting

The following functions support boot management.

BFROM_FINALINIT

Entry address:

0xEF00 0002

Arguments:

no arguments

C prototype:

```
void bfrom_FinalInit (void);
```

The `bfrom_FinalInit` function never returns. It only executes a `JUMP` to the address stored in `EVT1`.

BFROM_PDMA

Entry address:

0xEF00 0004

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_PDma (ADI_BOOT_DATA *p);
```

This is the load function for peripherals such as SPI and UART that support DMA in their boot modes.

BFROM_MDMA

Entry address:

0xEF00 0006

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_MDma (ADI_BOOT_DATA *p);
```

This is the load function used for memory boot modes. This routine is also reused when the `BFLAG_FILL` or the `BFLAG_INDIRECT` flags are specified.

BFROM_SPIBOOT

Entry address:

0xEF00 000A

Arguments:

- SPI address in R0
- dFlags in R1
- dBlockCount in R2
- pCallHook passed over the stack in [FP+0x14]
- updated block count returned in R0

C prototype:

```
s32 bfrom_SpiBoot (
    s32 dSpiAddress,
    s32 dFlags,
    s32 dBlockCount,
    ADI_BOOT_HOOK_FUNC* pCallHook);
```

This SPI master boot routine processes boot streams residing in SPI memories, using the SPI1 controller. The fourth argument pCallHook is passed over the stack. It provides a hook to call a callback routine after the ADI_BOOT_DATA structure is filled with default values. For example, the pCallHook routine may overwrite the default value of the uwSsel value in the ADI_BOOT_DATA structure. The coding follows the rules of uwHWAIT (see "Boot Host Wait (HWAIT) Feedback Strobe" on page 16-18). A value of 0x070B represents GPIO PG11 ($\overline{\text{SPI1\_SSEL5}}$).

Additional bits in the dFlags word are relevant. The user should always set the BFLAG_PERIPHERAL flag but never the BFLAG_SLAVE bit. The BFLAG_NOAUTO flag instructs the system to skip the SPI device detection

routine. The BFLAG_TYPE then tells the boot kernel what addressing mode is required for the SPI memory. (see "SPI Device Detection Routine" on page 16-42). The BFLAG_FASTREAD flag controls whether standard SPI read (0x3 command) or fast read (0xB) is performed. The three lower bits of the dFlags word are translated by the boot kernel into specific values to the SPI1_BAUD registers. This follows the truth table shown in Table 16-7 on page 16-42.

When called with the BFLAG_ALTERNATE flag, the bfrom_SpiBoot() function attempts to boot from external SPI memory device. Unless the uwSsel variable in the ADI_BOOT_DATA structure is altered by a hook routine, the memory is expected to be connected to $\overline{\text{SPI0\_SSEL2}}$. A pull-up resistor on this signal is required when automatic device detection is desired.

The bfrom_SpiBoot() routine does not deal with port muxing at all. When a part has been booted via SPI master mode after reset, the port muxing configuration is typically already ready for a runtime call to the bfrom_SpiBoot() routine. Otherwise ensure that the SPIx_MISO, SPIx_MOSI and SPIx_SCK signals are properly activated in the PORTx_FER and PORTx_MUX registers. The $\overline{\text{SPI0\_SSEL2}}$ signal requires, however, that the respective PORTx_FER bit be cleared, as the boot kernel toggles the signal in GPIO mode.

Similarly, the user shall set the PG11 bit in the PORTF_FER register when booting from an external device.

The bfrom_SpiBoot() routine uses the MDMA0 memory DMA channel pair and the DMA7 peripheral DMA. Respective wake-up bits must be set in the SIC_IWRx registers. If a different peripheral DMA channel has been assigned to the SPI0 controller, use the hook routine to store the MMR address of the respective DMAx_CONFIG register into the pDmaControlRegister variable in the ADI_BOOT_DATA structure. Similarly, when using a different SPI controller than SPI0, write the MMR address of the relevant SPIx_CTL register into the pControlRegister variable.

BFROM_BOOTKERNEL

Entry address:

0xEF00 0020

Arguments:

- pointer to `ADI_BOOT_DATA` in `R0`
- returns updated source address `pSource` in `R0`

C prototype:

```
s32 bfrom_BootKernel (
    ADI_BOOT_DATA *p);
```

This ROM entry provides access to the raw boot kernel routine. It is the user's responsibility to initialize the items passed in the `ADI_BOOT_DATA` structure. Pay particular attention that the function pointers (`pLoadFunction`, and `pErrorFunction`) point to functional routines.

BFROM_CRC32

Entry address:

0xEF00 0030

Arguments:

- pointer to look-up table in `R0`
- pointer to data in `R1`
- `dByteCount` in `R2`
- initial CRC value in `R0`
- CRC value returned in `R0`

C prototype:

```
s32 bfrom_Crc32 (
    s32 *pLut,
    void *pData,
    s32 dByteCount,
    s32 dInitial);
```

This routine calculates the CRC32 checksum for a given array of bytes. The look-up table is typically generated by the `BFROM_CRC32POLY` routine. During the boot process this routine is called by the `BFROM_CRC32CALLBACK` routine. The `dInitial` value is normally set to zero unless the CRC32 routine is called in multiple slices. Then, the `dInitial` parameter expects the result of the former run.

BFROM_CRC32POLY

Entry address:

0xEF00 0032

Arguments:

- pointer to look-up table in `R0`
- polynomial in `R1`
- updated block count returned in `R0`

C prototype:

```
s32 bfrom_Crc32Poly (
    unsigned s32 *pLut,
    s32 dPolynomial);
```

This function generates a 1024-byte look-up table from a given CRC polynomial. During the boot process this routine is hidden by the `BFROM_CRC32INITCODE` routine.

BFROM_CRC32CALLBACK

Entry address:

0xEF00 0034

Arguments:

- pointer to `ADI_BOOT_DATA` in `R0`
- pointer to `ADI_BOOT_BUFFER` in `R1* Callback Flags in R2`

C prototype:

```
s32 bfrom_Crc32Callback (
    ADI_BOOT_DATA *pBS,
    ADI_BOOT_BUFFER *pCS,
    s32 dCbFlags);
```

This is a wrapper function that ensures the `BFROM_CRC32` subroutine fits into the boot process.

BFROM_CRC32INITCODE

Entry address:

0xEF00 0036

Arguments:

pointer to

`ADI_BOOT_DATA` in `R0`

C prototype:

```
void bfrom_Crc32Initcode (
    ADI_BOOT_DATA *p);
```

This is an initcode residing in ROM with the following jobs:

- Register `BFROM_CRC32CALLBACK` as a callback routine to the `pCallback` pointer in `ADI_BOOT_DATA`.
- Call `BFROM_CRC32POLY` to generate the look-up table.

This function is unlikely to be called by user code directly. This function is called as an initcode during the boot process when the CRC calculation is desired. See "CRC Checksum Calculation" on page 16-30 for details.

Programming Examples

The following programming examples demonstrate various booting scenarios.

System Reset

To perform a system reset, use the code shown in Listing 16-1 or Listing 16-2.

Listing 16-1. System Reset in Assembly

```
#include <blackfin.h>
P0.L = LO(BFROM_SYSCONTROL);
P0.H = HI(BFROM_SYSCONTROL);
R0.L = LO(SYSCTRL_SYSRESET);
R0.H = HI(SYSCTRL_SYSRESET);
R1 = 0;
R2 = 0;
CALL (P0);
```

Listing 16-2. System Reset in C Language

```
bfrom_SysControl(
    SYSCTRL_SYSRESET,
    0,
    NULL);
```

Exiting Reset to User Mode

To exit reset while remaining in user mode, use the code shown in Listing 16-3.

Listing 16-3. Exiting Reset to User Mode

```
_reset:   P1.L = LO(_usercode);
            /* Point to start of user code */
     P1.H = HI(_usercode);
RETI = P1;  /* Load address of _start into RETI */
   RTI;     /* Exit reset priority */
_reset.end:
_usercode:  /* Place user code here */
...
```

The reset handler most likely performs additional tasks not shown in the examples above. Stack pointers and `EVTx` registers are initialized here.

Exiting Reset to Supervisor Mode

To exit reset while remaining in supervisor mode, use the code shown in Listing 16-4.

Listing 16-4. Exiting Reset by Staying in Supervisor Mode

```
_reset:
   P0.L = LO(EVT15);
  /* Point to IVG15 in Event Vector Table */
   P0.H = HI(EVT15);
   P1.L = LO(_isr_IVG15); /* Point to start of IVG15 code */
   P1.H = HI(_isr_IVG15);
   [P0] = P1;          /* Initialize interrupt vector EVT15 */
   P0.L = LO(IMASK);   /* read-modify-write IMASK register */
   R0 = [P0];          /* to enable IVG15 interrupts */
   R1 = EVT_IVG15 (Z);
   R0 = R0 | R1;      /* set IVG15 bit */
   [P0] = R0;         /* write back to IMASK */
   RAISE 15;          /* generate IVG15 interrupt request */
                      /* IVG 15 is not served until reset
                         handler returns */
   P0.L = LO(_usercode);
   P0.H = HI(_usercode);
   RETI = P0;         /* RETI loaded with return address */
   RTI;               /* Return from Reset Event */
_reset.end:
_usercode:             /* Wait in user mode till IVG15 */
   JUMP _usercode;     /* interrupt is serviced */
_isr_IVG15:            /* IVG15 vectors here due to EVT15 */
...
```

Initcode (Power Management Control)

The following examples show how to change PLL and the voltage regulator within an initcode.

The ADSP-BF59x processors do not have an on-chip voltage regulator. Set the `bfrom_SysControl` option to `SYSCTRL_EXTVOLTAGE`.

Listing 16-5. Changing PLL and Voltage Regulator in C Language

```
#include <ccblkfn.h>
#include <bfrom.h>
void init_DPM(ADI_BOOT_DATA* pBS)
{
    ADI_SYSCTRL_VALUES init_DPM;
    init_DPM.uwPllCtl = SET_MSEL(12);
    init_DPM.uwPllDiv = (SET_SSEL(4) | CSEL_DIV1);
    init_DPM.uwPllLockCnt = 0x0200;
    bfrom_SysControl(SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL |
    SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT | SYSCTRL_WRITE, &init_DPM,
NULL);
}
```

Listing 16-6. Changing PLL and Voltage Regulator in Assembly

```
#include <blackfin.h>
#include <bfrom.h>
.import "bfrom.h";
/* Load Immediate 32-bit value into data or address register */
#define IMM32(reg,val) reg##.H=hi(val); reg##.L=lo(val)
.SECTION L1_code;
init_DPM:
link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:5);
SP += -12;
R0.L = SET_MSEL(12);
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllCtl)] = R0;
R0.L = (SET_SSEL(4) | CSEL_DIV1);
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllDiv)] = R0;
R0.L = 0x0200;
```

```
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+offse-
tof(ADI_SYSCTRL_VALUES,uwPllLockCnt)] = R0;
R0 = (SYSCTRL_EXTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_LOCKCNT | SYSCTRL_WRITE);
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P5,BFROM_SYSCONTROL);
call(P5);
SP += 12;
(R7:0,P5:5) = [SP++];
unlink;
rts;
init_DPM.end:
```

Care must be taken that the reprogramming of the PLL does not break the communication with the booting host. For example, in the case of UART boot, the `UARTx_DLL` and `UARTx_DLH` registers must be updated to keep the old bit rate.

XOR Checksum

Listing 16-7 illustrates how an initcode can be used to register a callback routine. The routine is called after each boot block that has the `BFLAG_CALLBACK` flag set. The calculated XOR checksum is compared against the block header `ARGUMENT` field. When the checksum fails, this example goes into idle mode. Otherwise control is returned to the boot kernel.

Since this callback example accesses the data after it is loaded, it would fail if the target address were in L1 instruction space. Therefore the `BFLAG_INDIRECT` flag should also be set. The `xor_callback` routine could then perform the checksum calculation at an intermediate storage place. The boot kernel transfers the data from the temporary buffer to the final destination after the callback routine returns.

In general, the block size is bigger than the size of the temporary buffer. Therefore, the boot kernel may need to divide the processing of a single block into multiple steps. The callback routine may also need to be invoked multiple times—every time the temporary buffer is filled up and once for the remaining bytes. The boot kernel passes the `dFlags` parameter, so that the callback routines knows whether it is called the first time, the last time or neither. The `dUserLong` variable in the `ADI_BOOT_DATA` structure is used to store the intermediate results between function calls.

Listing 16-7. XOR Checksum

```
s32 xor_callback(
    ADI_BOOT_DATA* pBS,
    ADI_BOOT_BUFFER* pCS,
    s32 dFlags)
{
    s32 i;
    if ((pCS!= NULL) && (pBS->pHeader!= NULL)) {
        if (dFlags & CBFLAG_FIRST) {
            pBS->dUserLong = 0;
        }
        for (i=0; i<pCS->dByteCount/sizeof(s32); i++) {
            pBS->dUserLong^= ((s32 *)pCS->pSource)[i];
        }
        if (dFlags & CBFLAG_FINAL) {
            if (pBS->dUserLong!= pBS->pHeader->dArgument) {
                idle ();
            }
        }
    }
    return 0;
}
void xor_initcode (ADI_BOOT_DATA *pBS)
{
```

```
    pBS->pCallBackFunction = xor_callback;
}
```

Note that the callback routine is not volatile. It should not be overwritten by subsequent boot blocks. It can, however, be overwritten after processing the last block with `BFLAG_CALLBACK` flag set.

The checksum algorithm must be booted first and cannot protect itself. The ADSP-BF59x processors provide a CRC32 checksum algorithm in the on-chip L1 instruction ROM, that can be used for booting under this scenario. For more information see "CRC Checksum Calculation" on page 16-30.

17 SYSTEM DESIGN

This chapter provides hardware, software and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to the processor data sheet for pin information, including pin numbers.

Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor's `CLKIN` pin. It is not possible to halt, change, or operate `CLKIN` below the specified frequency during normal operation. The processor uses the clock input (`CLKIN`) to generate on-chip clocks. These include the core clock (`CCLK`) and the peripheral clock (`SCLK`).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKIN pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see Chapter 16, “Dynamic Power Management”.

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the system interrupt assignment registers (SIC_IARx). For more information, see Chapter 4, “System Interrupts”.

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, please see the *Blackfin Processor Programming Reference*.

Data Delays, Latencies and Throughput

For detailed information on latencies and performance estimates on the DMA and external memory buses, refer to “Chip Bus Hierarchy” on page 3-1.

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to "Chip Bus Hierarchy" on page 3-1.

High-Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Signal Integrity

In addition to reducing signal length and capacitive loading, critical signals should be treated like transmission lines.

Capacitive loading and signal length of buses can be reduced by using a buffer for devices that operate with wait states. This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes.

Use simple signal integrity methods to prevent transmission line reflections that may cause extraneous extra clock and sync signals. Additionally, avoid overshoot and undershoot that can cause long term damage to input pins.

Some signals are especially critical for short trace length and usually require series termination. The `CLKIN` pin should have impedance matching series resistance at its driver. SPORT interface signals `TCLK`, `RCLK`, `RFS`, and `TFS` should use some termination. Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances the drivers often require resistive termination located at

the source. (Note also that TFS and RFS should not be shorted in multi-channel mode.) On the PPI interface, the PPI_CLK and SYNC signals also benefit from these standard signal integrity techniques. If these pins have multiple sources, it will be difficult to keep the traces short.

Adding termination to fix a problem on an existing board requires delays for new artwork and new boards. A transmission line simulator is recommended for critical signals. IBIS models are available from Analog Devices Inc. that will assist signal simulation software. Some signals can be corrected with a small zero or 22 ohm resistor located near the driver. The resistor value can be adjusted after measuring the signal at all endpoints.

For details, see the reference sources in “Recommended Reading” on page 17-13 for suggestions on transmission line termination.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the Printed Circuit Board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes.
- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.
- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the VDDEXT and VDDINT pins of the package as shown in Figure 17-4. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground plane

inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

Connect the power plane to the power supply pins directly with minimum trace length. A ground plane should be located near the component side of the board to reduce the distance that ground current must travel through vias. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced.

VDDINT is the highest frequency and requires special attention. Two things help power filtering above 100 MHz. First, capacitors should be physically small to reduce the inductance. Surface mount capacitors of size 0402 give better results than larger sizes. Secondly, lower values of capacitance will raise the resonant frequency of the LC circuit. While a cluster of 0.1µF is acceptable below 50 MHz, a mix of 0.1µF, 0.01µF, 0.001µF and even 100 pF is preferred in the 500 MHz range.

Note that the instantaneous voltage on both internal and external power pins must at all times be within the recommended operating conditions as specified in the product data sheet. Local “bulk capacitance” (many microfarads) is also necessary. Although all capacitors should be kept close to the power consuming device, small capacitance values should be the closest and larger values may be placed further from the chip.

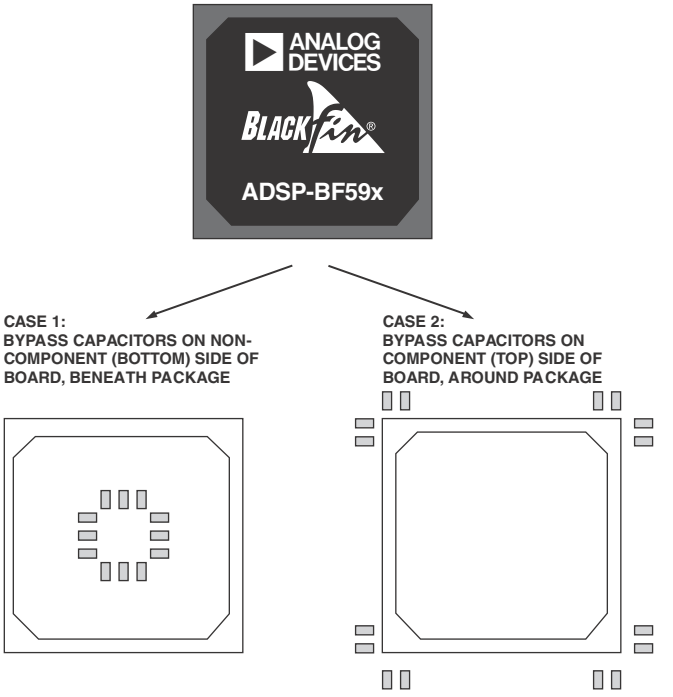

ANALOG DEVICES
BLACKFIN
ADSP-BF59x
CASE 1:
BYPASS CAPACITORS ON NON-COMPONENT (BOTTOM) SIDE OF BOARD, BENEATH PACKAGE
CASE 2:
BYPASS CAPACITORS ON COMPONENT (TOP) SIDE OF BOARD, AROUND PACKAGE

Figure 17-1. Bypass Capacitor Placement

Test Point Access

The debug process is aided by test points on signals such as CLKOUT, bank selects, PPICLK, and $\overline{\text{RESET}}$. If selection pins such as boot mode are connected directly to power or ground, they are inaccessible under the chip. Use pull-up and pull-down resistors instead.

Oscilloscope Probes

When making high-speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with

4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

Recommended Reading

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

This book is a technical reference that covers the problems encountered in state of the art, high-frequency digital circuit design. It is an excellent

source of information and practical ideas. Topics covered in the book include:

- High-speed properties of logic gates
- Measurement techniques
- Transmission lines
- Ground planes and layer stacking
- Terminations
- Vias
- Power systems
- Connectors
- Ribbon cables
- Clock distribution
- Clock oscillators

Consult your CAD software tools vendor. Some companies offer demonstration versions of signal integrity software. Simply by using their free software, you can learn:

- Transmission lines are real
- Unterminated printed circuit board traces will ring and have overshoot and undershoot
- Simple termination will control signal integrity problems

Resetting the Processor

The reset pin requires a monotonic rise and fall. Therefore the pin should not be connected directly to an R/C time delay because such a circuit could be noise sensitive. In addition to the hardware reset mode provided via the RESET pin, the processor supports several software reset modes. For detailed information on the various modes, see *Blackfin Processor Programming Reference*. The processor state after reset is also described in the programming reference.

Recommendations for Unused Pins

Most often, there is no need to terminate unused pins, but the handful that do require termination are listed at the end of the pin list description section of the product data sheet.

If the real-time clock is not used, RTXI should be pulled low.

Also note that unused peripherals may have separate power connections. These should be driven to the specified value.

Programmable Outputs

During power up, each GPIO pin is set to an input and any pins used in the system as an output should be connected to a pullup or pulldown resistor to maintain the desired state.

This would be particularly important in motor drive applications. It is also important for UART TX and RTS, SPI and serial TWI, or other communications interfaces. Some memory enable pull-ups may also be desired.

After the boot cycle, each GPIO pin may be set to input or output depending on ADSP-BF59x model number and the boot cycle chosen. The I/O / GPIO muxing of all pins may need to be reprogrammed to support the users application. Care should be taken for compatibility of function and state, before boot, during boot, and application pin usage.

Voltage Regulation Interface

ADSP-BF59x processors must use an external voltage regulator to power the V_{DDINT} domain. The EXT_WAKE and $\overline{\text{PG}}$ signals can facilitate communication with the external voltage regulator. EXT_WAKE is high-true for power-up and low only when the processor is in the hibernate state. EXT_WAKE may be connected directly to the low-true shut down input of many common regulators.

The $\overline{\text{PG}}$ (power-good, low-true) signal that allows the processor to start only after the internal voltage has reached a chosen level. In this way, the startup time of the external regulator will be detected after hibernation.

If the processor never will enter the hibernate state, the $\overline{\text{PG}}$ signal can be grounded in this mode. This will always indicate 'power good', meaning that V_{DDINT} is at a safe operating level. Any delay required at initial power-on, to guarantee a safe operating level for V_{DDINT}, will be provided by the RESET signal.

If the external regulator for V_{DDINT} has a power-good signal output, it can be used to help the processor recover properly from it's hibernate state. This signal may need to be inverted, as the processor's input should be low-true in order to indicate a "power good" condition.

If the external regulator does not have a power-good output, the $\overline{PG}$ signal should be driven to a fixed level (just below the desired operating voltage) so that the $\overline{PG}$ pin voltage can be compared to V_{DDINT} by the internal startup logic. This can be accomplished with an external resistor divider from V_{DDEXT} or any other fixed stable voltage. A divider with impedance of 1M Ohm is sufficient to supply current to this $\overline{PG}$ input. To save even more current during hibernation, the EXT_WAKE signal may be used as the voltage source to the divider. EXT_WAKE is low during hibernation, but will go high before the V_{DDINT} voltage is applied by the external regulator. In all cases, care should be taken to account for the min and max values of V_{DDEXT} or V_{OH} for EXT_WAKE. The voltage applied to the $\overline{PG}$ pin is used as the threshold that is compared internally to the rising value of V_{DDINT} to signal the processor to start. The voltage at $\overline{PG}$ should be calculated such that the V_{DDINT} value has risen to the desired voltage range for the application.

A SYSTEM MMR ASSIGNMENTS

This appendix lists MMR addresses and register names for all system registers. Table A-1 groups the registers by function/peripheral and indicates the section later in this chapter where individual registers for that group are listed. The tables in the later sections cross reference to individual register diagrams located in the chapter where that register is described. The diagrams show individual bit descriptions for each register.

Table A-1. Register Tables in This Chapter

Function/Peripheral
"System Reset and Interrupt Control Registers" on page A-3
"DMA/Memory DMA Control Registers" on page A-4
"Ports Registers" on page A-7
"Timer Registers" on page A-9
"Core Timer Registers" on page A-3
"Watchdog Timer Registers" on page A-11
"Dynamic Power Management Registers" on page A-11
"Processor-Specific Memory Registers" on page A-2
"PPI Registers" on page A-12
"SPI Controller Registers" on page A-12
"SPORT Controller Registers" on page A-14
"SPORT Clock Gating Register" on page A-17
"UART Controller Registers" on page A-18
"TWI Registers" on page A-19

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFDF FFFF.
- All system MMRs are either 16 bits or 32 bits wide. MMRs that are 16 bits wide must be accessed with 16-bit read or write operations. MMRs that are 32 bits wide must be accessed with 32-bit read or write operations. Check the description of the MMR to determine whether a 16-bit or a 32-bit access is required.
- All system MMR space that is not defined in this appendix is reserved for internal use only.

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in Table A-2.

Table A-2. Processor-Specific Memory Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFE0 0300	DTEST_COMMAND	"Data Test Command Register" on page 2-5

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in Table A-3.

Table A-3. Core Timer Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFE0 3000	TCNTL	"Core Timer Control Register (TCNTL)" on page 9-5
0xFFE0 3004	TPERIOD	"Core Timer Period Register (TPERIOD)" on page 9-6
0xFFE0 3008	TSCALE	"Core Timer Scale Register (TSCALE)" on page 9-7
0xFFE0 300C	TCOUNT	"Core Timer Count Register (TCOUNT)" on page 9-5

System Reset and Interrupt Control Registers

System reset and interrupt control registers (0xFFC0 0100 – 0xFFC0 01FF) are listed in Table A-4.

Table A-4. System Reset and Interrupt Control Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0104	SYSCR	"System Reset Configuration (SYSCR) Register" on page 16-54
0xFFC0 010C	SIC_IMASK0	"System Interrupt Mask (SIC_IMASK) Register" on page 4-12
0xFFC0 0110	SIC_IAR0	"System Interrupt Assignment (SIC_IAR) Register" on page 4-10

Table A-4. System Reset and Interrupt Control Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0114	SIC_IAR1	"System Interrupt Assignment (SIC_IAR) Register" on page 4-10
0xFFC0 0118	SIC_IAR2	"System Interrupt Assignment (SIC_IAR) Register" on page 4-10
0xFFC0 011C	SIC_IAR3	"System Interrupt Assignment (SIC_IAR) Register" on page 4-10
0xFFC0 0120	SIC_ISR0	"System Interrupt Status (SIC_ISR) Register" on page 4-12
0xFFC0 0124	SIC_IWR0	"System Interrupt Wakeup-Enable (SIC_IWR) Register" on page 4-12

DMA/Memory DMA Control Registers

DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in Table A-5.

Table A-5. DMA Traffic Control Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0B0C	DMA_TC_PER	"DMA_TC_PER Register" on page 5-88
0xFFC0 0B10	DMA_TC_CNT	"DMA_TC_CNT Register" on page 5-88

Since each DMA channel has an identical MMR set, with fixed offsets from the base address associated with that DMA channel, it is convenient to view the MMR information as provided in Table A-6 and Table A-7. Table A-6 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table A-7 then lists the register suffix and provides its offset from the Base Address.

As an example, the DMA channel 0 Y_MODIFY register is called `DMA0_Y_MODIFY`, and its address is 0xFFC0 0C1C. Likewise, the memory DMA stream 0 source current address register is called `MDMA_S0_CURR_ADDR`, and its address is 0xFFC0 0E64.

Table A-6. DMA Channel Base Addresses

DMA Channel Identifier	MMR Base Address	Register Prefix
0	0xFFC0 0C00	DMA0_
1	0xFFC0 0C40	DMA1_
2	0xFFC0 0C80	DMA2_
3	0xFFC0 0CC0	DMA3_
4	0xFFC0 0D00	DMA4_
5	0xFFC0 0D40	DMA5_
6	0xFFC0 0D80	DMA6_
7	0xFFC0 0DC0	DMA7_
8	0xFFC0 0E00	DMA8_
MemDMA stream 0 destination	0xFFC0 0F00	MDMA_D0_
MemDMA stream 0 source	0xFFC0 0F40	MDMA_S0_
MemDMA stream 1 destination	0xFFC0 0F80	MDMA_D1_
MemDMA stream 1 source	0xFFC0 0FC0	MDMA_S1_

Table A-7. DMA Register Suffix and Offset

Register Suffix	Offset From Base	For individual bits, see this diagram:
NEXT_DESC_PTR	0x00	"DMA Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/ MDMA_yy_NEXT_DESC_PTR)" on page 5-80
START_ADDR	0x04	"DMA Start Address Registers (DMAx_START_ADDR/MDMA_yy_START_ADDR)" on page 5-74

Table A-7. DMA Register Suffix and Offset (Continued)

Register Suffix	Offset From Base	For individual bits, see this diagram:
CONFIG	0x08	"DMA Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)" on page 5-67
X_COUNT	0x10	"DMA Inner Loop Count Registers (DMAx_X_COUNT/MDMA_yy_X_COUNT)" on page 5-75
X_MODIFY	0x14	"DMA Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)" on page 5-77
Y_COUNT	0x18	"DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)" on page 5-78
Y_MODIFY	0x1C	"DMA Outer Loop Address Increment Registers (DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY)" on page 5-79
CURR_DESC_PTR	0x20	"DMA Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)" on page 5-81
CURR_ADDR	0x24	"DMA Current Address Registers (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR)" on page 5-74
IRQ_STATUS	0x28	"DMA Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)" on page 5-71
PERIPHERAL_MAP	0x2C	"DMA Peripheral Map Registers DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP)" on page 5-66
CURR_X_COUNT	0x30	"DMA Current Inner Loop Count Registers (DMAx_CURR_X_COUNT /MDMA_yy_CURR_X_COUNT)" on page 5-76
CURR_Y_COUNT	0x38	"DMA Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)" on page 5-78

Ports Registers

Ports registers (port F: 0xFFC0 0700 – 0xFFC0 07FF, port G: 0xFFC0 1500 – 0xFFC0 15FF, port H: 0xFFC0 1700 – 0xFFC0 17FF, pin control: 0xFFC0 3200 – 0xFFC0 32FF) are listed in Table A-8.

Table A-8. Ports Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0700	PORTFIO	"GPIO Data Registers" on page 7-23
0xFFC0 0704	PORTFIO_CLEAR	"GPIO Clear Registers" on page 7-24
0xFFC0 0708	PORTFIO_SET	"GPIO Set Registers" on page 7-24
0xFFC0 070C	PORTFIO_TOGGLE	"GPIO Toggle Registers" on page 7-25
0xFFC0 0710	PORTFIO_MASKA	"GPIO Mask Interrupt A Registers" on page 7-27
0xFFC0 0714	PORTFIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 7-30
0xFFC0 0718	PORTFIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 7-28
0xFFC0 071C	PORTFIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 7-32
0xFFC0 0720	PORTFIO_MASKB	"GPIO Mask Interrupt B Registers" on page 7-27
0xFFC0 0724	PORTFIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 7-31
0xFFC0 0728	PORTFIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 7-29
0xFFC0 072C	PORTFIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 7-33
0xFFC0 0730	PORTFIO_DIR	"GPIO Direction Registers" on page 7-22
0xFFC0 0734	PORTFIO_POLAR	"GPIO Polarity Registers" on page 7-25

Table A-8. Ports Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0738	PORTFIO_EDGE	"Interrupt Sensitivity Registers" on page 7-26
0xFFC0 073C	PORTFIO_BOTH	"GPIO Set on Both Edges Registers" on page 7-26
0xFFC0 0740	PORTFIO_INEN	"GPIO Input Enable Registers" on page 7-23
0xFFC0 1500	PORTGIO	"GPIO Data Registers" on page 7-23
0xFFC0 1504	PORTGIO_CLEAR	"GPIO Clear Registers" on page 7-24
0xFFC0 1508	PORTGIO_SET	"GPIO Set Registers" on page 7-24
0xFFC0 150C	PORTGIO_TOGGLE	"GPIO Toggle Registers" on page 7-25
0xFFC0 1510	PORTGIO_MASKA	"GPIO Mask Interrupt A Registers" on page 7-27
0xFFC0 1514	PORTGIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 7-30
0xFFC0 1518	PORTGIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 7-28
0xFFC0 151C	PORTGIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 7-32
0xFFC0 1520	PORTGIO_MASKB	"GPIO Mask Interrupt B Registers" on page 7-27
0xFFC0 1524	PORTGIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 7-31
0xFFC0 1528	PORTGIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 7-29
0xFFC0 152C	PORTGIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 7-33
0xFFC0 1530	PORTGIO_DIR	"GPIO Direction Registers" on page 7-22
0xFFC0 1534	PORTGIO_POLAR	"GPIO Polarity Registers" on page 7-25
0xFFC0 1538	PORTGIO_EDGE	"Interrupt Sensitivity Registers" on page 7-26

Table A-8. Ports Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 153C	PORTGIO_BOTH	"GPIO Set on Both Edges Registers" on page 7-26
0xFFC0 1540	PORTGIO_INEN	"GPIO Input Enable Registers" on page 7-23
0xFFC0 3200	PORTF_FER	"Function Enable Registers" on page 7-22
0xFFC0 3204	PORTG_FER	"Function Enable Registers" on page 7-22
0xFFC0 3210	PORTF_MUX	"Port Multiplexer Control Register" on page 7-21
0xFFC0 3214	PORTG_MUX	"Port Multiplexer Control Register" on page 7-21

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in Table A-9.

Table A-9. Timer Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0600	TIMER0_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 8-40
0xFFC0 0604	TIMER0_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 8-41
0xFFC0 0608	TIMER0_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 8-42
0xFFC0 060C	TIMER0_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 8-42
0xFFC0 0610	TIMER1_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 8-40

Table A-9. Timer Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0614	TIMER1_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 8-41
0xFFC0 0618	TIMER1_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 8-42
0xFFC0 061C	TIMER1_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 8-42
0xFFC0 0620	TIMER2_CONFIG	"Timer Configuration Register (TIMER_CONFIG)" on page 8-40
0xFFC0 0624	TIMER2_COUNTER	"Timer Counter Register (TIMER_COUNTER)" on page 8-41
0xFFC0 0628	TIMER2_PERIOD	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 8-42
0xFFC0 062C	TIMER2_WIDTH	"Timer Period (TIMER_PERIOD) and Timer Width (TIMER_WIDTH) Registers" on page 8-42
0xFFC0 0680	TIMER_ENABLE	"Timer Enable Register (TIMER_ENABLE)" on page 8-35
0xFFC0 0684	TIMER_DISABLE	"Timer Disable Register (TIMER_DISABLE)" on page 8-36
0xFFC0 0688	TIMER_STATUS	"Timer Status Register (TIMER_STATUS)" on page 8-38

Watchdog Timer Registers

Watchdog timer registers (0xFFC0 0200 – 0xFFC0 02FF) are listed in Table A-10.

Table A-10. Watchdog Timer Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0200	WDOG_CTL	"Watchdog Control (WDOG_CTL) Register" on page 10-7
0xFFC0 0204	WDOG_CNT	"Watchdog Count (WDOG_CNT) Register" on page 10-5
0xFFC0 0208	WDOG_STAT	"Watchdog Status (WDOG_STAT) Register" on page 10-6

Dynamic Power Management Registers

Dynamic power management registers (0xFFC0 0000 – 0xFFC0 00FF) are listed in Table A-11.

Table A-11. Dynamic Power Management Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0000	PLL_CTL	"PLL_CTL Register" on page 6-20
0xFFC0 0004	PLL_DIV	"PLL_DIV Register" on page 6-19
0xFFC0 0008	VR_CTL	"VR_CTL Register" on page 6-21
0xFFC0 000C	PLL_STAT	"PLL_STAT Register" on page 6-20
0xFFC0 0010	PLL_LOCKCNT	"PLL_LOCKCNT Register" on page 6-21

PPI Registers

PPI registers (0xFFC0 1000 – 0xFFC0 10FF) are listed in Table A-12.

Table A-12. PPI Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1000	PPI_CONTROL	"PPI Control Register (PPI_CONTROL)" on page 15-25
0xFFC0 1004	PPI_STATUS	"PPI Status Register (PPI_STATUS)" on page 15-29
0xFFC0 1008	PPI_COUNT	"PPI Transfer Count Register (PPI_COUNT)" on page 15-32
0xFFC0 100C	PPI_DELAY	"PPI Delay Count Register (PPI_DELAY)" on page 15-32
0xFFC0 1010	PPI_FRAME	"PPI Lines Per Frame Register (PPI_FRAME)" on page 15-33

SPI Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in Table A-13.

SPI1 controller registers (0xFFC0 3400 – 0xFFC0 34FF) are listed in Table A-14.

Table A-13. SPI0 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0500	SPI0_CTL	"SPI Control (SPI_CTL) Register" on page 13-35
0xFFC0 0504	SPI0_FLG	"SPI Flag (SPI_FLG) Register" on page 13-37
0xFFC0 0508	SPI0_STAT	"SPI Status (SPI_STAT) Register" on page 13-38

Table A-13. SPI0 Controller Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 050C	SPI0_TDBR	"SPI Transmit Data Buffer (SPI_TDBR) Register" on page 13-41
0xFFC0 0510	SPI0_RDBR	"SPI Receive Data Buffer (SPI_RDBR) Register" on page 13-42
0xFFC0 0514	SPI0_BAUD	"SPI Baud Rate (SPI_BAUD) Register" on page 13-34
0xFFC0 0518	SPI0_SHADOW	"SPI RDBR Shadow (SPI_SHADOW) Register" on page 13-43

Table A-14. SPI1 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 3400	SPI1_CTL	"SPI Control (SPI_CTL) Register" on page 13-35
0xFFC0 3404	SPI1_FLG	"SPI Flag (SPI_FLG) Register" on page 13-37
0xFFC0 3408	SPI1_STAT	"SPI Status (SPI_STAT) Register" on page 13-38
0xFFC0 340C	SPI1_TDBR	"SPI Transmit Data Buffer (SPI_TDBR) Register" on page 13-41
0xFFC0 3410	SPI1_RDBR	"SPI Receive Data Buffer (SPI_RDBR) Register" on page 13-42
0xFFC0 3414	SPI1_BAUD	"SPI Baud Rate (SPI_BAUD) Register" on page 13-34
0xFFC0 3418	SPI1_SHADOW	"SPI RDBR Shadow (SPI_SHADOW) Register" on page 13-43

SPORT Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in Table A-15. SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in Table A-16.

Table A-15. SPORT0 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0800	SPORT0_TCR1	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 14-46
0xFFC0 0804	SPORT0_TCR2	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 14-46
0xFFC0 0808	SPORT0_TCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 14-61
0xFFC0 080C	SPORT0_TFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 14-62
0xFFC0 0810	SPORT0_TX	"SPORT Transmit Data (SPORT_TX) Register" on page 14-56
0xFFC0 0818	SPORT0_RX	"SPORT Receive Data (SPORT_RX) Register" on page 14-58
0xFFC0 0820	SPORT0_RCR1	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 14-51
0xFFC0 0824	SPORT0_RCR2	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 14-51
0xFFC0 0828	SPORT0_RCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 14-61
0xFFC0 082C	SPORT0_RFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 14-62
0xFFC0 0830	SPORT0_STAT	"SPORT Status (SPORT_STAT) Register" on page 14-60

Table A-15. SPORT0 Controller Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0834	SPORT0_CHNL	"SPORT Current Channel (SPORT_CHNL) Register" on page 14-64
0xFFC0 0838	SPORT0_MCMC1	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 14-63
0xFFC0 083C	SPORT0_MCMC2	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 14-63
0xFFC0 0840	SPORT0_MTCS0	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 0844	SPORT0_MTCS1	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 0848	SPORT0_MTCS2	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 084C	SPORT0_MTCS3	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 0850	SPORT0_MRCS0	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65
0xFFC0 0854	SPORT0_MRCS1	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65
0xFFC0 0858	SPORT0_MRCS2	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65
0xFFC0 085C	SPORT0_MRCS3	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65

Table A-16. SPORT1 Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0900	SPORT1_TCR1	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 14-46
0xFFC0 0904	SPORT1_TCR2	"SPORT Transmit Configuration (SPORT_TCR1 and SPORT_TCR2) Registers" on page 14-46

Table A-16. SPORT1 Controller Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0908	SPORT1_TCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 14-61
0xFFC0 090C	SPORT1_TFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 14-62
0xFFC0 0910	SPORT1_TX	"SPORT Transmit Data (SPORT_TX) Register" on page 14-56
0xFFC0 0918	SPORT1_RX	"SPORT Receive Data (SPORT_RX) Register" on page 14-58
0xFFC0 0920	SPORT1_RCR1	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 14-51
0xFFC0 0924	SPORT1_RCR2	"SPORT Receive Configuration (SPORT_RCR1 and SPORT_RCR2) Registers" on page 14-51
0xFFC0 0928	SPORT1_RCLKDIV	"SPORT Transmit and Receive Serial Clock Divider (SPORT_TCLKDIV and SPORT_RCLKDIV) Registers" on page 14-61
0xFFC0 092C	SPORT1_RFSDIV	"SPORT Transmit and Receive Frame Sync Divider (SPORT_TFSDIV and SPORT_RFSDIV) Registers" on page 14-62
0xFFC0 0930	SPORT1_STAT	"SPORT Status (SPORT_STAT) Register" on page 14-60
0xFFC0 0934	SPORT1_CHNL	"SPORT Current Channel (SPORT_CHNL) Register" on page 14-64
0xFFC0 0938	SPORT1_MCMC1	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 14-63
0xFFC0 093C	SPORT1_MCMC2	"SPORT Multichannel Configuration (SPORT_MCMC1 and SPORT_MCMC2) Registers" on page 14-63
0xFFC0 0940	SPORT1_MTCS0	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 0944	SPORT1_MTCS1	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66

Table A-16. SPORT1 Controller Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0948	SPORT1_MTCS2	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 094C	SPORT1_MTCS3	"SPORT Multichannel Transmit Selection (SPORT_MTCSn) Registers" on page 14-66
0xFFC0 0950	SPORT1_MRCS0	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65
0xFFC0 0954	SPORT1_MRCS1	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65
0xFFC0 0958	SPORT1_MRCS2	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65
0xFFC0 095C	SPORT1_MRCS3	"SPORT Multichannel Receive Selection (SPORT_MRCSn) Registers" on page 14-65

SPORT Clock Gating Register

The SPORT clock gating register (0xFFC0 120C) is listed in Table A-18.

Table A-17. SPORT Clock Gating Register

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 120C	SPORT_CLKGATE	"SPORT Clock Gating Register" on page 14-76

UART Controller Registers

UART controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in Table A-18.

Table A-18. UART Controller Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 0400	UART_THR	"UART Transmit Holding (UART_THR) Register" on page 11-25
0xFFC0 0400	UART_RBR	"UART Receive Buffer (UART_RBR) Register" on page 11-26
0xFFC0 0400	UART_DLL	"UART Divisor Latch (UART_DLL and UART_DLH) Registers" on page 11-29
0xFFC0 0404	UART_DLH	"UART Divisor Latch (UART_DLL and UART_DLH) Registers" on page 11-29
0xFFC0 0404	UART_IER	"UART Interrupt Enable (UART_IER) Register" on page 11-26
0xFFC0 0408	UART_IIR	"UART Interrupt Identification (UART_IIR) Register" on page 11-28
0xFFC0 040C	UART_LCR	"UART Line Control (UART_LCR) Register" on page 11-21
0xFFC0 0410	UART_MCR	"UART Modem Control (UART_MCR) Register" on page 11-23
0xFFC0 0414	UART_LSR	"UART Line Status (UART_LSR) Register" on page 11-24
0xFFC0 041C	UART_SCR	"UART Scratch (UART_SCR) Register" on page 11-30
0xFFC0 0424	UART_GCTL	"UART Global Control (UART_GCTL) Register" on page 11-31

TWI Registers

Two Wire Interface (TWI) registers (0xFFC0 1400 – 0xFFC0 14FF) are listed in Table A-19.

Table A-19. TWI Registers

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1400	TWI_CLKDIV	"SCL Clock Divider Register (TWI_CLKDIV)" on page 12-25
0xFFC0 1404	TWI_CONTROL	"TWI CONTROL Register (TWI_CONTROL)" on page 12-24
0xFFC0 1408	TWI_SLAVE_CTL	"TWI Slave Mode Control Register (TWI_SLAVE_CTL)" on page 12-26
0xFFC0 140C	TWI_SLAVE_STAT	"TWI Slave Mode Status Register (TWI_SLAVE_STAT)" on page 12-28
0xFFC0 1410	TWI_SLAVE_ADDR	"TWI Slave Mode Address Register (TWI_SLAVE_ADDR)" on page 12-28
0xFFC0 1414	TWI_MASTER_CTL	"TWI Master Mode Control Register (TWI_MASTER_CTL)" on page 12-29
0xFFC0 1418	TWI_MASTER_STAT	"TWI Master Mode Status Register (TWI_MASTER_STAT)" on page 12-33
0xFFC0 141C	TWI_MASTER_ADDR	"TWI Master Mode Address Register (TWI_MASTER_ADDR)" on page 12-32
0xFFC0 1420	TWI_INT_STAT	"TWI Interrupt Status Register (TWI_INT_STAT)" on page 12-40
0xFFC0 1424	TWI_INT_MASK	"TWI Interrupt Mask Register (TWI_INT_MASK)" on page 12-39
0xFFC0 1428	TWI_FIFO_CTL	"TWI FIFO Control Register (TWI_FIFO_CTL)" on page 12-36
0xFFC0 142C	TWI_FIFO_STAT	"TWI FIFO Status Register (TWI_FIFO_STAT)" on page 12-38
0xFFC0 1480	TWI_XMT_DATA8	"TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)" on page 12-43

Table A-19. TWI Registers (Continued)

Memory-Mapped Address	Register Name	For individual bits, see this diagram:
0xFFC0 1484	TWI_XMT_DATA16	"TWI FIFO Transmit Data Double Byte Register (TWI_XMT_DATA16)" on page 12-43
0xFFC0 1488	TWI_RCV_DATA8	"TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)" on page 12-44
0xFFC0 148C	TWI_RCV_DATA16	"TWI FIFO Receive Data Double Byte Register (TWI_RCV_DATA16)" on page 12-45

B TEST FEATURES

This appendix discusses the test features of the processor.

JTAG Standard

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards.The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a boundary-scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a boundary-scan register and other building blocks. The test logic is accessed through a Test Access Port (TAP).

Full details of the JTAG standard can be found in the document *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN 1-55937-350-4.

Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table B-1)
- A TAP controller that controls all sequencing of events through the test registers
- An instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

Table B-1. Test Access Port Pins

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.

Figure B-1 shows the state diagram for the TAP controller.

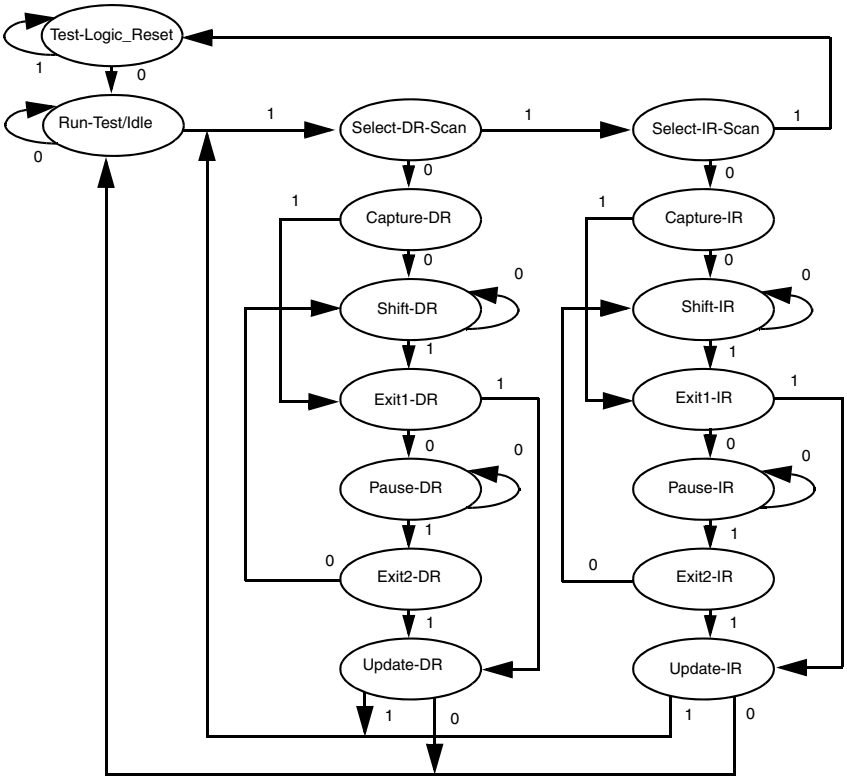


Figure B-1. TAP Controller State Diagram

Note:

- The TAP controller enters the test-logic-reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the test-logic-reset state when TRST is asynchronously asserted.
- An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

The binary decode column of Table B-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table B-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
SAMPLE/PRELOAD	10000	Boundary-Scan
BYPASS	11111	Bypass

Figure B-2 shows the instruction bit scan ordering for the paths shown in Table B-2.

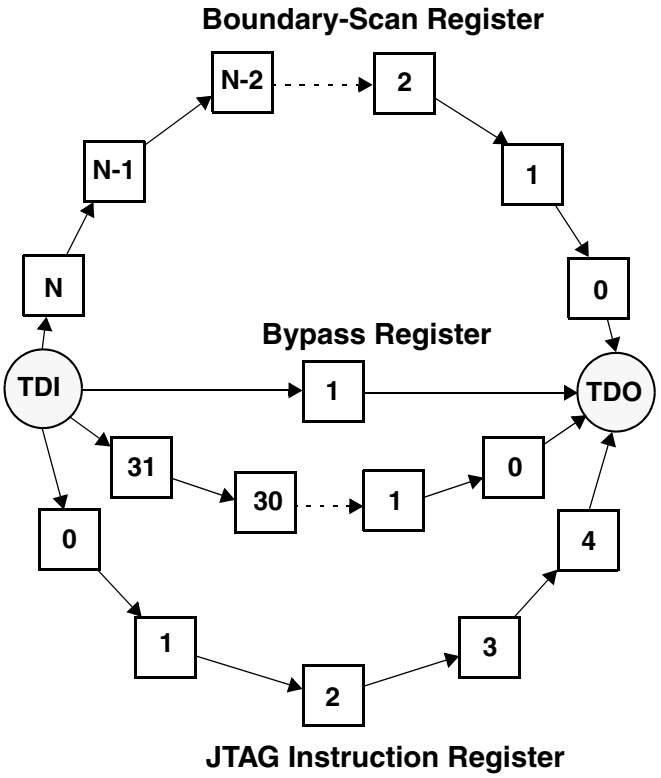


Figure B-2. Serial Scan Paths

Public Instructions

The following sections describe the public JTAG scan instructions.

EXTEST – Binary Code 00000

The EXTEST instruction selects the boundary-scan register to be connected between the TDI and TDO pins. This instruction allows testing of on-board circuitry external to the device.

The EXTEST instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

The SAMPLE part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of TCK.

The PRELOAD part of the instruction allows data to be loaded on the device pins and driven out on the board with the EXTEST instruction. Data is preloaded on the pins on the falling edge of TCK.

BYPASS – Binary Code 11111

The BYPASS instruction selects the BYPASS register to be connected to TDI and TDO. The instruction has no effect on the internal logic. No data inversion should occur between TDI and TDO.

Boundary-Scan Register

The boundary-scan register is selected by the EXTEST and SAMPLE/PRELOAD instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing.

I INDEX

Numerics

2D DMA, 5-11
2X input clock, 14-24

A

aborts, DMA, 5-28
accesses
  off-core, 3-3
  to internal memory, 2-1
access way•instruction address bit 11 bit, 2-5, 2-6
active descriptor queue, and DMA synchronization, 5-59
active low/high frame syncs, serial port, 14-32
active mode, 1-17, 6-8
ACTIVE_PLLDISABLED bit, 6-20
ACTIVE_PLLENABLED bit, 6-20
active video only mode, PPI, 15-10
ADCs, connecting to, 14-2
A-law companding, 14-23, 14-28
alternate frame sync mode, 14-35
alternate timing, serial port, 14-34
ANAK (address not acknowledged) bit, 12-33, 12-35
application data, loading, 16-1
arbitration
  DAB, 3-6
  DCB, 3-6
  TWI, 12-8
architecture, memory, 2-1
array access bit, 2-5, 2-6
asynchronous
  serial communications, 11-5
autobaud, and general-purpose timers, 8-31
autobaud detection, 11-13
autobuffer mode, 5-11, 5-28, 5-68

B

bandwidth, and memory DMA operations, 5-46
baud rate
  SPI, 13-34
  UART, 11-7, 11-13
baud rate[15:0] field, 13-34
BCINIT[15:0] field, 5-84
BCODE [3-0] field, 16-54
BCOUNT[15:0] field, 5-85
BDI (block done interrupt generated) bit, 5-83
BDIE (block done interrupt enable) bit, 5-40, 5-83
BFLAG_ALTERNATE bit, 16-63
BFLAG_AUX bit, 16-12, 16-64
BFLAG_CALLBACK bit, 16-12, 16-64
BFLAG_FASTREAD bit, 16-63
BFLAG_FILL bit, 16-12, 16-64
BFLAG_FINAL bit, 16-12, 16-64
BFLAG_FIRST bit, 16-64
BFLAG_first bit, 16-12
BFLAG_HDRINDIRECT bit, 16-63

BFLAG_HOOK bit, 16-63
BFLAG_IGNORE bit, 16-12, 16-64
BFLAG_INDIRECT bit, 16-12, 16-64
BFLAG_INIT bit, 16-12, 16-64
BFLAG_NEXTDXE bit, 16-63
BFLAG_NOAUTO bit, 16-63
BFLAG_NONRESTORE bit, 16-63
BFLAG_PERIPHERAL bit, 16-63
BFLAG_QUICKBOOT bit, 16-12, 16-64
BFLAG_RESET bit, 16-63
BFLAG_RETURN bit, 16-63
BFLAG_SAVE bit, 16-12, 16-64
BFLAG_SLAVE bit, 16-63
BFLAG_TYPE bit, 16-63
BFLAG_WAKEUP bit, 16-63
BI (break interrupt) bit, 11-24, 11-25
binary decode, B-4
bit order, selecting, 14-27
bit rate generation, 11-12
BKDATECODE (boot code dated code) register, 16-56
BK_DAY field, 16-56
BK_ID field, 16-55
BK_MONTH field, 16-56
BK_ONES (boot code ones) register, 16-58
BK_ONES field, 16-58
BK_PROJECT field, 16-55
BKREVISION (boot code revision) register, 16-55
BK_UPDATE field, 16-55
BK_VERSION field, 16-55
BK_YEAR field, 16-56
BK_ZERO field, 16-57
BKZEROS (boot code zeros) register, 16-57
Blackfin processor family
  memory architecture, 1-5
block, DMA, 5-9
block code field, 16-12
Block Code word, 16-12
block count, DMA, 5-37
block diagrams
  bus hierarchy, 3-2
  core, 3-3
  core timer, 9-2
  DMA controller, 5-102
  general-purpose timers, 8-57
  PLL, 6-3
  PPI, 15-3
  processor, 1-4
  SPI, 13-3, 13-4
  SPORT, 14-6
  TWI, 12-3
  UART, 11-3
  watchdog timer, 10-3
block done interrupt, DMA, 5-40
Block Flags, 16-14
block transfers, DMA, 5-37
BMODE[2:0] pins, 16-4
BMODE[2-0] field, 16-54
BMODE pins, 16-2
boot
  call boot kernel at run time, 16-32
  load function, 16-31
  manager, 16-35
  quick, 16-25
  ROM functions, 16-36
  streams
    multi-DXE, 16-36
boot code date code (BKDATECODE) register, 16-56
boot code ones (BK_ONES) register, 16-58
boot code revision (BKREVISION) register, 16-55
boot code zero word (BK_ZEROS) register, 16-57
boot host wait
  HWAIT, 16-18

booting, 16-1 to 16-71
  boot stream, 16-8
  host boot scenarios, 16-8
  indirect, 16-26
  initialization code execution/boot, 16-24
  memory locations, 16-9
  SPI slave mode, 16-44
booting modes, 16-2
boot kernel, 16-1
Boot Management, 16-35
boot mode
  no-boot, 16-39
  SPI device detection, 16-42
boot ROM
  internal, 16-1
boot stream, 16-1, 16-8
boot termination, 16-19
boundary-scan architecture, B-2
boundary-scan register, B-7
broadcast mode, 13-8, 13-14, 13-15
buffer registers, timers, 8-43
BUFRDERR (buffer read error) bit, 12-33, 12-35
BUFWRERR (buffer write error) bit, 12-33, 12-34
bus agents
  DAB, 3-7
  PAB, 3-5
BUSBUSY (bus busy) bit, 12-33
buses
  See also DAB, DCB, PAB
  bandwidth, 1-4
  core, 3-3
  hierarchy, 3-2
  on-chip, 3-1
  PAB, 3-4
  peripheral, 3-4
  and peripherals, 1-3
  prioritization and DMA, 5-48
bus standard, I^2C, 1-8
bypass
  capacitor placement, 17-6
BYPASS bit, 6-20
BYPASS instruction, B-6
BYPASS register, B-6

C

callback routines, 16-27
capacitors, 17-4
capture mode, *See* WDTH_CAP mode
CCIR-656, *See* ITU-R 656
CCITT G.711 specification, 14-28
CCLK (core clock), 6-5
  status by operating mode, 6-7
CCLK (core processor clock), 3-3
channels
  defined, serial, 14-22
  serial port TDM, 14-22
  serial select offset, 14-22
CHNL[9:0] field, 14-64, 14-65
circuit board testing, B-1, B-6
circular addressing, 5-56
clearing interrupt requests, 4-13
clear Pxn bit, 7-24
clear Pxn interrupt A enable bit, 7-30
clear Pxn interrupt B enable bit, 7-31
CLKHI[7:0] field, 12-26
CLKIN (input clock), 1-16, 3-3, 6-1, 6-2
CLKLOW[7:0] field, 12-26
CLK_SEL (timer clock select) bit, 8-12, 8-20, 8-41, 8-46
clock
  clock signals, 1-16
  control, 6-1
  external, 1-16
  frequency for SPORT, 14-61
  internal, 3-3
  managing, 17-1
  peripheral, 6-7
  source for general-purpose timers, 8-3

clock *(continued)*
SPI clock signal, 13-4
system, 1-16
system (SCLK), 17-2
types, 17-1
clock divide modulus registers, 14-61
clock domain synchronization, PPI, 15-15
clock input (CLKIN) pin, 17-1
clock phase, SPI, 13-11, 13-13
clock polarity, SPI, 13-11
clock rate
core timer, 9-1
SPORT, 14-2
clock ratio, changing, 6-6
clocks, overview, 1-16
clock signals, 1-16
codecs, connecting to, 14-2
commands
DMA control, 5-31, 5-32
transfer initiate, 13-17, 13-18
companding, 14-15, 14-23
defined, 14-28
lengths supported, 14-28
multichannel operations, 14-23
configuration
SPORT, 14-10
congestion, on DMA channels, 5-45
continuous transition, DMA, 5-26
control bit summary, general-purpose timers, 8-45
control byte sequences, PPI, 15-8
core
block diagram, 3-3
core bus, 3-3
core clock (CCLK), 6-5, 17-2
core clock/system clock ratio control, 6-5
timer, 4-5
waking from idle state, 4-6
core and system reset, code example, 16-71, 16-72
core clock, *See* CCLK
core clock (CCLK), 9-2
core double-fault reset, 16-4
core event controller (CEC), 4-2
core-only software reset, 16-4
core select (CSEL) bits, 6-19
core timer, 9-1 to 9-8
block diagram, 9-2
clock rate, 9-1
features, 9-2
initialization, 9-3
internal interfaces, 9-3
low power mode, 9-3
operation, 9-3
registers, 9-4
scaling, 9-7
core timer control (TCNTL) register, 9-3, 9-5
core timer count (TCOUNT) register, 9-3, 9-5
core timer scale (TSCALE) register, 9-3, 9-7
count value[15:0] field, 9-6
count value[31:16] field, 9-6
CPHA bit, 13-36
CPOL bit, 13-36
CRC32 checksum generation, 16-30
CROSSCORE software, 1-19
crosstalk, 17-4
crystal
external, 1-16
CSEL[1:0] field, 6-5, 6-19, 17-2
CTYPE (DMA channel type) bit, 5-66
current address field, 5-75
current address registers
(DMAx_CURR_ADDR), 5-74
(MDMA_yy_CURR_ADDR), 5-74
current descriptor pointer (DMAx_CURR_DESC_PTR) registers, 5-81

current descriptor pointer
(MDMA_yy_CURR_DESC_PTR)
registers, 5-81
current inner loop count registers
(DMAx_CURR_X_COUNT), 5-76
(MDMA_yy_CURR_X_COUNT), 5-76
current outer loop count registers
(DMAx_CURR_Y_COUNT), 5-78
(MDMA_yy_CURR_Y_COUNT), 5-78
CURR_X_COUNT[15:0] field, 5-76
CURR_Y_COUNT[15:0] field, 5-79
customer support, iv

D

DAB, 3-6, 5-5, 5-41, 5-89
arbitration, 3-6
bus agents (masters), 3-7
latencies, 3-8
performance, 3-8
DAB_TRAFFIC_COUNT[2:0] field, 5-89
data
sampling, serial, 14-32
data corruption, avoiding with SPI, 13-14
data CPLB data register
(DCPLB_DATAx), 2-8
data-driven interrupts, 5-72
data formats, SPORT, 14-27
data input modes for PPI, 15-14 to 15-16
data memory control register
(DMEM_CONTROL), 2-7
data move, serial port operations, 14-37
data output modes for PPI, 15-16 to 15-18
data structures, 16-58
boot_struct, 16-60
buffer_struct, 16-59
header_struct, 16-59
data test command register
(DTEST_COMMAND), 2-5
data transfers
DMA, 3-8, 5-2
SPI, 13-14
data word, serial data formats, 14-55
DCB, 3-6, 5-5, 5-41, 5-90
arbitration, 3-6
DCB_TRAFFIC_COUNT field, 5-90
DCB_TRAFFIC_PERIOD field, 5-90
DCNT[7:0] field, 12-29, 12-30
DCPLB_DATAx (data CPLB data)
register, 2-8
DEB_TRAFFIC_COUNT field, 5-89
DEB_TRAFFIC_PERIOD field, 5-89
debugging, 1-20
test point access, 17-6
deep sleep mode, 1-18, 6-9
delaycount (PPI_DELAY) register, 15-32
descriptor
array mode, DMA, 5-15, 5-68
chains, DMA, 5-26
list mode, DMA, 5-14, 5-68, 5-69
descriptor-based DMA, 5-13
descriptor queue, 5-57
management, 5-56
synchronization, 5-56, 5-57
descriptor structures
DMA, 5-55
MDMA, 5-62
destination channels, memory DMA, 5-7
development tools, 1-19
DF bit, 6-4, 6-20
DFETCH bit, 5-14, 5-21, 5-72
dFlags word, 16-63
DFRESET bit, 16-54
DI_EN bit, 5-14, 5-67, 5-69
direct code execution
initial header, 16-20
direct memory access, *See* DMA

disabling
PLL, 6-11
DI_SEL bit, 5-67, 5-69
DITFS (data-independent transmit frame sync select) bit, 14-36, 14-47, 14-50, 14-60
divisor latch high byte[15:8] field, 11-30
divisor latch low byte[7:0] field, 11-30
divisor reset, UART, 11-30
DLAB (divisor latch access) bit, 11-21, 11-26, 11-27
DLEN[2:0] field, 15-25, 15-26
DMA, 5-1 to 5-101
1-D interrupt-driven, 5-53
1-D unsynchronized FIFO, 5-55
2-D, polled, 5-54
2-D array, example, 5-90
2-D interrupt-driven, 5-53
autobuffer mode, 5-11, 5-28, 5-68
bandwidth, 5-45
block count, 5-37
block diagram, 5-102
block done interrupt, 5-40
block transfers, 5-9, 5-37
buffer size, multichannel SPORT, 14-23
buses, 3-6
channel registers, 5-65
channels, 5-41
channels and control schemes, 5-50
channel-specific register names, 5-64
congestion, 5-45
connecting asynchronous FIFO, 5-38
continuous transfers using autobuffering, 5-53
continuous transition, 5-26
control command restrictions, 5-34
control commands, 5-31, 5-32
controllers, 1-6
data transfers, 5-2
descriptor array, 5-22
DMA *(continued)*
descriptor array mode, 5-15, 5-68
descriptor-based, 5-13
descriptor-based, initializing, 5-93
descriptor-based vs. register-based transfers, 5-3
descriptor chains, 5-26
descriptor element offsets, 5-16
descriptor list mode, 5-14, 5-68, 5-69
descriptor lists, 5-22
descriptor queue, 5-56, 5-57
descriptors, recommended size, 5-16
descriptor structures, 5-55
direction, 5-70
DMA error interrupt, 5-72
double buffer scheme, 5-53
and EBIU, 5-4
errors, 5-28, 5-30
example connection, receive, 5-39
example connection, transmit, 5-38
external interfaces, 5-4
features, 5-2
finish control command, 5-33
first data memory access, 5-21
flow chart, 5-18, 5-19
FLOW mode, 5-16
FLOW value, 5-20
for SPI transmit, 13-9
handshake operation, 5-36
header file to define descriptor structures example, 5-94
HMDMA1 block enable example, 5-99
HMDMA with delayed processing example, 5-100
initializing, 5-17
internal interfaces, 5-4
and L1 memory, 5-5
large model mode, 5-69
latency, 5-24
memory conflict, 5-48

DMA *(continued)*
memory DMA, 1-7, 5-6
memory DMA streams, 5-7
memory DMA transfers, 5-5
memory read, 5-25
operation flow, 5-17
orphan access, 5-28
overflow interrupt, 5-40
overview, 1-6
performance considerations, 5-42
peripheral, 5-5
peripheral channels priority, 5-6
peripheral interrupts, 4-6
peripheral priority and default mapping, 5-103
pipelining requests, 5-37
polling DMA status example, 5-93
polling registers, 5-51
and PPI, 15-35
prioritization and traffic control, 5-44 to 5-49
programming examples, 5-90 to 5-101
receive, 5-26
refresh, 5-22
register-based, 5-9
register-based 2D memory DMA example, 5-91
register naming conventions, 5-65
remapping peripheral assignment, 5-6
request data control command, 5-34
request data urgent control command, 5-34
restart control command, 5-32, 5-33
round robin operation, 5-47
serial port block transfers, 14-37
single-buffer transfers, 5-52
small model mode, 5-68
software management, 5-50
software-triggered descriptor fetch example, 5-96

DMA *(continued)*
and SPI, 13-10
SPI data transmission, 13-41, 13-42
SPI master, 13-23
SPI slave, 13-26
and SPORT, 14-4
startup, 5-17
stop mode, 5-11, 5-68
stopping transfers, 5-28
support for peripherals, 1-4
switching peripherals from, 5-73
and synchronization with PPI, 15-13
synchronization, 5-50 to 5-60
synchronized transition, 5-27
termination without abort, 5-28
throughput, 5-41
traffic control, 5-48
traffic exceeding available bandwidth, 5-45
transfers, 1-6
transfers, urgent, 5-44
transmit, 5-25
transmit restart or finish, 5-34, 5-35
triggering transfers, 5-60
two descriptors in small list flow mode, example, 5-94
two-dimensional, 5-11
two-dimensional memory DMA setup example, 5-92
types supported, 1-6
and UART, 11-17, 11-27
using descriptor structures example, 5-95
variable descriptor size, 5-15
with PPI, 15-22
word size, changing, 5-27
work units, 5-14, 5-22, 5-24
DMA2D bit, 5-67, 5-70
DMA bus, *See* DAB
DMACFG field, 5-21, 5-61
DMA channel registers, 5-62

DMACODE field, 16-12, 16-64
DMA Code field
DMACODE, 16-12
DMA configuration (DMAx_CONFIG) registers, 5-67
DMA configuration (MDMA_yy_CONFIG) registers, 5-67
DMA controller, 5-2
DMA core bus, *See* DCB
DMA direction (WNR) bit, 5-67, 5-70
DMA_DONE bit, 5-10, 5-72
DMA_DONE interrupt, 5-71
DMAEN bit, 5-17, 5-60, 5-67, 5-70
DMA_ERR bit, 5-10, 5-72
DMA_ERROR interrupt, 5-29
DMA error interrupts, 5-72
DMA performance optimization, 5-40
DMA queue completion interrupt, 5-59
DMA registers, 5-62
DMA_RUN bit, 5-21, 5-58, 5-61, 5-71, 5-72
DMA_RUN bit), 5-10
DMARx pin, 5-37
DMA start address field, 5-74
DMA_TC_CNT (DMA traffic control counter) register, 5-88, 5-89
DMA_TC_PER (DMA traffic control counter period) register, 5-46, 5-88
DMA traffic control registers, 5-88
DMA_TRAFFIC_PERIOD field, 5-89
DMAx_CONFIG (DMA configuration) registers, 5-8, 5-17, 5-24, 5-67
DMAx_CURR_ADDR (current address) registers, 5-74
DMAx_CURR_DESC_PTR (current descriptor pointer) registers, 5-81
DMAx_CURR_X_COUNT (current inner loop count) registers, 5-76
DMAx_CURR_Y_COUNT (current outer loop count) registers, 5-78
DMAx_IRQ_STATUS (interrupt status) registers, 5-71, 5-72
DMAx_NEXT_DESC_PTR (next descriptor pointer) registers, 5-17, 5-80
DMAx_PERIPHERAL_MAP (peripheral map) registers, 4-6, 5-66
DMAx_START_ADDR (start address) registers, 5-17, 5-74
DMAx_X_COUNT (inner loop count) registers, 5-75
DMAx_X_MODIFY (inner loop address increment) registers, 5-17, 5-77
DMAx_Y_COUNT (outer loop count) registers, 5-78
DMAx_Y_MODIFY (outer loop address increment) registers, 5-17, 5-79
DMEM_CONTROL (data memory control) register, 2-7
DNAK (data not acknowledged) bit, 12-33, 12-35
DOUBLE_FAULT bit, 16-52
DPMC, 6-2, 6-7 to 6-18
DR (data ready) bit, 11-12, 11-24
DR flag, 11-16
DRQ[1:0] field, 5-45, 5-82, 5-83
DRxPRI signal, 14-5
DRxPRI SPORT input, 14-5
DRxSEC signal, 14-5
DRxSEC SPORT input, 14-5
DSP libraries, 1-21
DTEST_COMMAND (data test command) register, 2-5
DTxPRI signal, 14-5
DTxPRI SPORT output, 14-5
DTxSEC signal, 14-5
DTxSEC SPORT output, 14-5

dynamic power management, 1-16, 6-1
  controller, 6-2

E

early frame sync, 14-34
EAV signal, 15-5
EBIU
  and DMA, 5-4
ECINIT[15:0] field, 5-86
ECOUNT[15:0] field, 5-86
edge detection, GPIO, 7-12
elfloader.exe, 16-8
ELSI (enable Rx status interrupt) bit, 11-7, 11-27, 11-28
EMISO (enable MISO) bit, 13-35, 13-36
emulation, and timer counter, 8-42
EMU_RUN bit, 8-41, 8-42, 8-46
enable Pxn interrupt A bit, 7-27
enable Pxn interrupt B bit, 7-27
enabling
  interrupts, 4-5
entire field mode, PPI, 15-9
EPS (even parity select) bit, 11-21
ERBFI (enable receive buffer full interrupt) bit, 11-7, 11-12, 11-26, 11-27
ERR_DET (error detected) bit, 15-29, 15-30
ERR_NCOR (error not corrected) bit, 15-30
errors
  DMA, 5-28
  not detected by DMA hardware, 5-30
  startup, and timers, 8-8
error signals, SPI, 13-38 to 13-41
ERR_TYP[1:0] field, 8-7, 8-40, 8-41, 8-46
ERR_TYP bits, 8-28
ETBEI (enable transmit buffer empty interrupt) bit, 11-6, 11-11, 11-17, 11-26, 11-27
event controller, 4-2
event handling, 4-2
events
  definition, 4-3
  types of, 4-2
event system, 4-3
event vector table (EVT), 4-2
EVT1 register, 16-6
EXT_CLK mode, 8-32, 8-43
  control bit and register usage, 8-45
  flow diagram, 8-33
external
  emulator debugger, 8-42
external crystal, 1-16
EXTEST instruction, B-6
EZ-KIT Lite card, 1-21

F

FAST (fast mode) bit, 12-29, 12-31
fast mode, TWI, 12-10
FE (framing error) bit, 11-24, 11-25
FFE (force framing error on transmit) bit, 11-31
FIFO
  asynchronous connection, 5-38
finish control command, DMA, 5-33
FLD (field indicator) bit, 15-30, 15-31
FLD_SEL (active field select) bit, 15-4, 15-26, 15-29
flex descriptors, 5-3
FLGx (slave select value) bit, 13-37, 13-38
FLOW[2:0] field, 5-22, 5-23, 5-55, 5-67, 5-68
flow charts
  DMA, 5-18, 5-19
  general-purpose timers interrupt structure, 8-6
  GPIO, 7-17
  GPIO interrupt generation, 7-14
  PPI, 15-24
  SPI core-driven, 13-29

flow charts *(continued)*
SPI DMA, 13-30
timer EXT_CLK mode, 8-33
timer PWM_OUT mode, 8-11
timer WDTH_CAP mode, 8-23
TWI master mode, 12-23
TWI slave mode, 12-22
FLOW mode, DMA, 5-16
FLOW (next operation) bit, 5-14, 5-15
FLOW value, DMA, 5-20
FLSx (slave select enable) bit, 13-8, 13-37
FPE bit, 11-31
framed serial transfers, characteristics, 14-31
framed/unframed data, 14-30
frame start detect, PPI, 15-34
frame sync
active high/low, 14-32
early, 14-34
early/late, 14-34
external/internal, 14-31
internal, 14-25
internally generated, 14-62
late, 14-34
multichannel mode, 14-18
sampling edge, 14-32
SPORT options, 14-30
frame sync divider[15:0] field, 14-63
frame synchronization
PPI in GP modes, 15-19
and SPORT, 14-3
frame sync polarity, PPI and timer, 15-20
frame sync pulse
use of, 14-50
when issued, 14-50
frame sync signal, control of, 14-49, 14-54
frame track error, 15-30, 15-33
frequencies, clock and frame sync, 14-25
FSDR (frame sync to data relationship) bit, 14-21, 14-64
F signal, 15-31
FT_ERR (frame track error) bit, 15-30, 15-33
full duplex, 14-4, 14-5
SPI, 13-2
FULL_ON bit, 6-20
full-on mode, 1-17, 6-8
function enable (PORTF_FER) register, 7-6
function enable (PORTG_FER) register, 7-6
function enable (PORTH_FER) register, 7-6
function enable (PORTx_FER) registers, 7-22

G

GCALL (general call) bit, 12-28
general call address, TWI, 12-10
general-purpose interrupts, 4-2, 4-3
general-purpose I/O, *See* GPIO
general-purpose ports, 7-1 to 7-34
assigning interrupt channels, 7-13
interrupt channels, 7-12, 7-13
interrupt generation flow, 7-12
latency, 7-7
pin defaults, 7-2
pins, interrupt, 7-11
throughput, 7-6
general-purpose ports, See GPIO
general-purpose timers, 8-1 to 8-56
aborting immediately, 8-22
and startup errors, 8-8
autobaud mode, 8-31
block diagram, 8-57
buffer registers, 8-43
capture mode, 8-5
clock source, 8-3
code examples, 8-47
control bit summary, 8-45

general-purpose timers *(continued)*
counter, 8-4
disable timing, 8-22
enabling, 8-5, 8-33
error detection, 8-7
EXT_CLK mode, 8-43
external interface, 8-3
features, 8-2
flow diagram for EXT_CLK mode, 8-33
generating maximum frequency, 8-15
illegal states, 8-7, 8-9
internal interface, 8-4
internal timer structure, 8-3
interrupts, 8-4, 8-5, 8-14, 8-28
interrupt setup, 8-49
interrupt structure, 8-6
measurement report, 8-25, 8-26, 8-27
non-overlapping clock pulses, 8-53
output pad disable, 8-12
overflow, 8-4
periodic interrupt requests, 8-50
port setup, 8-47
and PPI, 8-58
preventing errors in PWM_OUT mode, 8-44
programming model, 8-33
PULSE_HI toggle mode, 8-15
PWM mode, 8-5
PWM_OUT mode, 8-10 to 8-23, 8-43
registers, 8-34
signal generation, 8-48
single pulse generation, 8-12
size of register accesses, 8-35
stopping in PWM_OUT mode, 8-21
three timers with same period, 8-17
two timers with non-overlapping clocks, 8-17
waveform generation, 8-13
WDTH_CAP mode, 8-23, 8-43
WDTH_CAP mode configuration, 8-55
general-purpose timers *(continued)*
WDTH_CAP mode flow diagram, 8-23
GEN (general call enable) bit, 12-26, 12-27
glitch filtering, UART, 11-9
GM (get more data) bit, 13-20, 13-36
GPIO, 1-7, 7-1 to 7-34
assigned to same interrupt channel, 7-15
clearing interrupt conditions, 7-13
clear registers, 7-10
code examples, 7-33
configuration, 7-8
data registers, 7-8, 7-9, 7-10
direction registers, 7-8, 7-12
edge detection, 7-12
edge-sensitive, 7-9
flow chart, 7-17
function enable registers, 7-7, 7-11
input buffers, 7-8
input driver, 7-9
input drivers, 7-12
input enable registers, 7-8, 7-11
interrupt channels, 7-16
interrupt generation flow chart, 7-14
interrupt request, 4-14
interrupts, 7-12
interrupt sensitivity registers, 7-12
mask data registers, 7-13
mask interrupt clear registers, 7-15
mask interrupt set registers, 7-13
mask interrupt toggle registers, 7-15
mask registers, 7-13
pins, 7-7, 7-8
polarity registers, 7-11
registers, 7-20
set registers, 7-10
toggle registers, 7-11
using as input, 7-8
write operations, 7-9
writes to registers, 7-10

GPIO clear (PORTxIO_CLEAR) registers, 7-24
GPIO data (PORTxIO) registers, 7-23
GPIO direction (PORTxIO_DIR) registers, 7-22
GPIO input enable (PORTxIO_INEN) registers, 7-23
GPIO mask interrupt A clear registers, 7-30
GPIO mask interrupt A (PORTxIO_MASKA_CLEAR) registers, 7-27
GPIO mask interrupt A set (PORTxIO_MASKA_SET) registers, 7-28
GPIO mask interrupt A toggle (PORTxIO_MASKA_TOGGLE) registers, 7-32
GPIO mask interrupt B clear (PORTxIO_MASKB_CLEAR) registers, 7-31
GPIO mask interrupt B (PORTxIO_MASKB) registers, 7-27
GPIO mask interrupt B set (PORTxIO_MASKB_SET) registers, 7-29
GPIO mask interrupt B toggle (PORTxIO_MASKB_TOGGLE) registers, 7-33
GPIO pins, 7-7
GPIO polarity (PORTxIO_POLAR) registers, 7-25
GPIO set on both edges (PORTxIO_BOTH) registers, 7-26
GPIO set (PORTxIO_SET) registers, 7-24
GPIO toggle (PORTxIO_TOGGLE) registers, 7-25
GP modes, PPI, 15-14
ground plane, 17-4, 17-5

H

H.100, 14-21
H.100 standard protocol, 14-24
handshake MDMA, 5-8
  interrupts, 5-39
handshake MDMA configuration (HMDMAx_BCINIT) registers, 5-36
handshake MDMA control (HMDMAx_CONTROL) registers, 5-82
handshake MDMA control registers, 5-83
handshake MDMA current block count (HMDMAx_BCOUNT) registers, 5-37, 5-84
handshake MDMA current block count registers (HMDMAx_BCOUNT), 5-85
handshake MDMA current edge count (HMDMAx_ECOUNT) registers, 5-37, 5-38, 5-85, 5-86
handshake MDMA edge count overflow interrupt (HMDMAx_ECOVERFLOW) registers, 5-87
handshake MDMA edge count urgent (HMDMAx_ECURGENT) registers, 5-87
handshake MDMA initial block count (HMDMAx_BCINIT) registers, 5-84
handshake MDMA initial edge count (HMDMAx_ECINIT) registers, 5-38, 5-86
handshaking MDMA operation, 5-4
handshaking memory DMA (HMDMA), 5-2
hardware reset, 16-3, 16-4, 16-6
HDRCHK field, 16-12
HDRSGN field, 16-12
header checksum field
  HDRCHK, 16-15

HIBERNATEB bit, 6-17
hibernate state, 1-18, 6-10
high-frequency design considerations, 17-3
HMDMA, 5-2, 5-8
HMDMAEN bit, 5-36, 5-38, 5-83
HMDMAx_BCINIT (handshake MDMA configuration) registers, 5-36, 5-84
HMDMAx_BCOUNT (handshake MDMA current block count) registers, 5-37, 5-84, 5-85
HMDMAx_CONTROL (handshake MDMA control) registers, 5-82, 5-83
HMDMAx_ECINIT (handshake MDMA initial edge count) registers, 5-38, 5-86
HMDMAx_ECOUNT (handshake MDMA current edge count) registers, 5-37, 5-38, 5-85, 5-86
HMDMAx_ECOVERFLOW (handshake MDMA edge count overflow interrupt) registers, 5-87
HMDMAx_ECURGENT (handshake MDMA edge count urgent) registers, 5-87
HMVIP, 14-24
horizontal blanking, 15-6
horizontal tracking, PPI, 15-31

I

I2C, *See* TWI
I^2C bus standard, 1-8, 12-2
I^2S, 1-11
  format, 14-10
  serial devices, 14-3
ICPLB_DATAx (instruction CPLB data) register, 2-9
idle state
  waking from, 4-6
IEEE 1149.1 standard, *See* JTAG standard
IMASK (interrupt mask) register
  initialization, 4-8
IMEM_CONTROL (instruction memory control) register, 2-7
INIT bit, 16-22
initcall address/symbol command, 16-23
initcode routines, 16-21
initialization
  IMASK register, 4-8
  interrupt, 4-7
initializing
  DMA, 5-17
init initcode.dxe command, 16-23
inner loop address increment registers
  (DMAx_X_MODIFY), 5-77
  (MDMA_yy_X_MODIFY), 5-77
inner loop count registers
  (DMAx_X_COUNT), 5-75
  (MDMA_yy_X_COUNT), 5-75
input buffers, GPIO, 7-8
input clock, *See* CLKIN
input driver, GPIO, 7-9
instruction bit scan ordering, B-5
instruction CPLB data register (ICPLB_DATAx), 2-9
instruction memory control register (IMEM_CONTROL), 2-7
instruction register (IR), B-2, B-4
instructions, 1-18
  private, B-4
  public, B-4
  See also instructions by name
instruction test command register (ITEST_COMMAND), 2-6
interfaces
  on-chip, 3-2
  overview, 3-2
  system, 3-1
inter IC bus, 12-2
interlaced video, 15-6

interleaving
of data in SPORT FIFO, 14-56
SPORT data, 14-5
internal
clocks, 3-3
internal boot ROM, 16-1
internal/external frame syncs, *See* frame sync
internal memory, 1-5
accesses, 2-1
internal TSR register, UART, 11-6
interrupt
for peripheral, 4-1
interrupt conditions, UART, 11-28
interrupt handler and DMA synchronization, 5-58
interrupt output, SPI, 13-16
interrupt request lines, peripheral, 4-15
interrupts, 4-1 to 4-15
assigning priority for UART, 11-12
channels, assigning, 7-13
channels, GPIO, 7-12
clearing requests, 4-13
configuring and servicing, 17-2
control of system, 4-2
default mapping, 4-3
definition, 4-3
determining source, 4-5
DMA channels, 4-6
DMA_ERROR, 5-29
DMA error, 5-72
DMA overflow, 5-40
DMA queue completion, 5-59
enabling, 4-5
evaluation of GPIO interrupts, 7-15
general-purpose, 4-2, 4-3
general-purpose timers, 8-4, 8-5, 8-14, 8-28
generated by peripherals, 4-8
GPIO, 7-11, 7-13, 7-16
interrupts *(continued)*
handshake MDMA, 5-39
initialization, 4-7
inputs and outputs, 4-4
mapping, 4-4
mask function, 4-7
multiple sources, 4-10
peripheral, 4-2, 4-3, 4-4 to 4-7
prioritization, 4-4
processing, 4-1, 4-8
programming examples, 4-13 to 4-15
reset, 16-6
routing overview, 4-16
shared, 4-4
software, 4-3
SPI, 13-16, 13-46
SPORT error, 14-37
SPORT RX, 14-37, 14-59
SPORT TX, 14-37, 14-57
system, 4-1
to wake core from idle, 4-6
UART, 11-11
use in managing a descriptor queue, 5-57
interrupt sensitivity (PORTxIO_EDGE) registers, 7-26
interrupt service routine, determining source of interrupt, 4-5
interrupt status registers
(DMAx_IRQ_STATUS), 5-71, 5-72
(MDMA_yy_IRQ_STATUS), 5-71, 5-72
I/O interface to peripheral serial device, 14-4
I/O memory space, 1-5
I/O pins, general-purpose, 7-8
IRCLK (internal receive clock select) bit, 14-52, 14-54
IrDA, 11-31
receiver, 11-9
transmitter, 11-8

IrDA SIR, 11-5
IREN (enable IdDA mode) bit, 11-31
IRFS (internal receive frame sync select) bit, 14-31, 14-52, 14-54
IR instruction register, B-2, B-4
IRPOL bit, 11-10
IRQ bit, 8-47
IRQ_ENA bit, 8-41, 8-45, 8-47
ISR
  supporting multiple interrupt sources, 4-7, 4-17
ISR and multiple interrupt sources, 4-10
ITCLK (internal transmit clock select) bit, 14-47, 14-49
ITEST_COMMAND (instruction test command) register, 2-6
ITFS (internal transmit frame sync select) bit, 14-19, 14-31, 14-47, 14-50
ITHR[15:0] field, 5-87
ITU-R 601/656, 1-9
ITU-R 601 recommendation, 15-16
ITU-R 656 modes, 15-5, 15-9, 15-29
  active video only submode, 15-9, 15-10
  and DLEN field, 15-25
  entire field submode, 15-9
  frame start detect, 15-34
  frame synchronization, 15-11
  output, 15-11
  SAV codes, 15-31
  supported, 1-10
  vertical blanking interval only submode, 15-9, 15-10

J

JTAG, 1-21, B-1, B-2, B-4

L

L1
  data memory, 1-5
L1 *(continued)*
  data memory subbanks, 2-3
  data SRAM, 2-3
  instruction memory, 1-5, 2-2
  memory and core, 3-3
  memory and DMA controller, 5-5
  scratchpad RAM, 1-5
L1 instruction memory
  address alignment, 2-2
  subbanks, 2-2
LARFS (late receive frame sync) bit, 14-34, 14-52, 14-55
large descriptor mode, DMA, 5-14
large model mode, DMA, 5-69
late frame sync, 14-17, 14-34
latency
  DAB, 3-8
  DMA, 5-24
  general-purpose ports, 7-7
LATFS (late transmit frame sync) bit, 14-34, 14-47, 14-51
lines per frame (PPI_FRAME) register, 15-34
lines per frame register, 15-33
line terminations, SPORT, 14-8
little endian byte order, 12-43
loader file, 16-8
loader utility, 16-8
LOCKCNT[15:0] field, 6-21
locked transfers, DMA, 3-8
loopback feature, PPI, 15-10
loopback mode, UART, 11-23
LOOP (loopback mode enable) bit, 11-23
LOSTARB (lost arbitration) bit, 12-33, 12-35
LRFS (low receive frame sync select) bit, 14-12, 14-31, 14-32, 14-52, 14-55
LSBF (LSB first) bit, 13-36
LT_ERR_OVR flag, 15-31

LT_ERR_OVR (horizontal tracking overflow error) bit, 15-30, 15-31
LT_ERR_UNDR flag, 15-32
LT_ERR_UNDR (horizontal tracking underflow error) bit, 15-30, 15-31
LTFS (low transmit frame sync select) bit, 14-19, 14-31, 14-32, 14-47, 14-50

M

MADDR[6:0] field, 12-32
masters
  DAB, 3-7
  PAB, 3-5
MBDI bit, 5-40, 5-83
MCCRM[1:0] field, 14-64
MCDRXPE (multichannel DMA receive packing) bit, 14-64
MCDTXPE (multichannel DMA transmit packing) bit, 14-64
MCMEN (multichannel frame mode enable) bit, 14-17, 14-64
MCOMP (master transfer complete) bit, 12-40, 12-41
MCOMPM (master transfer complete interrupt mask) bit, 12-40
MDIR (master transfer direction) bit, 12-29, 12-31
MDMA channels, 5-6
MDMA controllers, 5-6
MDMA_ROUND_ROBIN_COUNT[4:0] field, 5-47, 5-88
MDMA_ROUND_ROBIN_PERIOD field, 5-46, 5-47, 5-88
MDMA_yy_CONFIG (DMA configuration) registers, 5-67
MDMA_yy_CURR_ADDR (current address) registers, 5-74
MDMA_yy_CURR_DESC_PTR (current descriptor pointer) registers, 5-81
MDMA_yy_CURR_X_COUNT (current inner loop count) registers, 5-76
MDMA_yy_CURR_Y_COUNT (current outer loop count) registers, 5-78
MDMA_yy_IRQ_STATUS (interrupt status) registers, 5-71, 5-72
MDMA_yy_NEXT_DESC_PTR (next descriptor pointer) registers, 5-80
MDMA_yy_PERIPHERAL_MAP (peripheral map) registers, 5-66
MDMA_yy_START_ADDR (start address) registers, 5-74
MDMA_yy_X_COUNT (inner loop count) registers, 5-75
MDMA_yy_X_MODIFY (inner loop address increment) registers, 5-77
MDMA_yy_Y_COUNT (outer loop count) registers, 5-78
MDMA_yy_Y_MODIFY (outer loop address increment) registers, 5-79
measurement report, general-purpose timers, 8-25, 8-26, 8-27
memory, 2-1 to 2-5
  accesses to internal, 2-1
  architecture, 1-5, 2-1
  boot ROM, 2-4
  configurations, 1-5
  internal, 1-5
  L1, 3-3
  L1 data, 1-5, 2-3
  L1 instruction, 1-5, 2-2
  L1 scratchpad RAM, 1-5
  moving data between SPORT and, 14-37
  on-chip, 1-5
  start locations of L1 instruction memory subbanks, 2-2
  structure, 1-4
memory conflict, DMA, 5-48

memory DMA, 1-7, 5-6
  bandwidth, 5-43
  buffers, 5-8
  channels, 5-7
  descriptor structures, 5-62
  handshake operation, 5-8
  priority, 5-46
  scheduling, 5-46
  timing, 5-44
  transfer operation, starting, 5-8
  transfers, 5-2, 5-5
  word size, 5-7
memory map
  ADSP-BF51x, 2-2
memory-mapped registers, *See* MMRs
memory-to-memory transfers, 5-7
MEN (master mode enable) bit, 12-29, 12-31
MERR (master transfer error) bit, 12-40, 12-41
MERRM (master transfer error interrupt mask) bit, 12-40
MFD[3:0] field, 14-20, 14-64
MISO pin, 13-5, 13-12, 13-14, 13-15, 13-20
MMRs, 1-5
  address range, A-2
  for PPI, 15-25
  memory-related, 2-4
  width, A-2
mode fault error, 13-16, 13-39
modes
  broadcast, 13-8, 13-14, 13-15
  multichannel, 14-14
  serial port, 14-10
  SPI master, 13-14, 13-17
  SPI slave, 13-15, 13-19
  UART DMA, 11-17
  UART non-DMA, 11-15
MODF (mode fault error) bit, 13-39
MOSI pin, 13-5, 13-12, 13-14, 13-15, 13-20
moving data, serial port, 14-37
MPROG (master transfer in progress) bit, 12-33, 12-35
MSEL[5:0] field, 6-4, 6-20
MSTR (master) bit, 13-35, 13-36
multichannel frame, 14-19
multichannel frame delay field, 14-20
multichannel mode, 14-14
  enable/disable, 14-17
  frame syncs, 14-18
  SPORT, 14-18
multichannel operation, SPORT, 14-14 to 14-24
multiple interrupt sources, 4-10
multiple slave SPI systems, 13-8
multiplexing, 7-1
MVIP-90, 14-24

N

NAK (not acknowledge) bit, 12-26, 12-27
NDPH bit, 5-20
NDPL bit, 5-20
NDSIZE[3:0] field, 5-15, 5-67, 5-69
  legal values, 5-31
next descriptor pointer registers
  (DMAx_NEXT_DESC_PTR), 5-80
  (MDMA_yy_NEXT_DESC_PTR), 5-80
nFlags variable, 16-63
NINT (pending interrupt) bit, 11-28, 11-29
normal frame sync mode, 14-34
normal timing, serial port, 14-34
NTSC systems, 15-6

O

OE (overrun error) bit, 11-24
off-chip
  infrared driver, 11-9
  line drivers, 11-7
  peripherals, 5-2
  signals, 7-11
off-core
  accesses, 3-3
offsets, DMA descriptor elements, 5-16
OI bit, 5-83
OIE bit, 5-83
on-chip
  busses, 3-6
  I/O devices, 1-5
  memory, 1-5
  peripherals, 1-6, 5-2
  PLL, 1-16
open drain drivers, 13-2
open drain outputs, 13-14
operating modes, 6-7
  active, 1-17, 6-8
  deep sleep, 1-18, 6-9
  full-on, 1-17, 6-8
  hibernate state, 1-18, 6-10
  PPI, 15-4
  sleep, 1-17, 6-9
  transition, 6-10, 6-12
optimization, of DMA performance, 5-40
oscilloscope probes, 17-6
OUT_DIS bit, 8-40, 8-41, 8-46, 8-58
outer loop address increment registers
  (DMAx_Y_MODIFY), 5-79
  (MDMA_yy_Y_MODIFY), 5-79
outer loop count registers
  (DMAx_Y_COUNT), 5-78
  (MDMA_yy_Y_COUNT), 5-78
output pad disable, timer, 8-12
overflow interrupt, DMA, 5-40

P

PAB, 3-4
  arbitration, 3-5
  bus agents (masters, slaves), 3-5
  clocking, 6-2
  performance, 3-6
PACK_EN (packing mode enable) bit, 15-26, 15-27
packing, serial port, 14-23
PAL systems, 15-6
parallel peripheral interface, *See* PPI
PDWN bit, 6-20
PEN (parity enable) bit, 11-21
PE (parity error) bit, 11-24
performance
  DAB, 3-8
  DCB, 3-8
  DMA, 5-42
  general-purpose ports, 7-6
  memory DMA, 5-43
  optimization, DMA, 5-40
  PAB, 3-6
PERIOD_CNT bit, 8-12, 8-20, 8-24, 8-25, 8-41, 8-45
period value[15:0] field, 9-6
period value[31:16] field, 9-6
peripheral
  DMA, 5-5
  DMA channels, 5-41
  DMA transfers, 5-2
  error interrupts, 5-72
  interrupt request lines, 4-15
  supporting interrupts, 4-1
peripheral access bus, *See* PAB
Peripheral bus
  errors generated by SPORT, 14-38
peripheral DMA start address registers, 5-74
peripheral interrupts, 4-2, 4-3, 4-4 to 4-7

peripheral map registers
(DMAx_PERIPHERAL_MAP), 5-66
(MDMA_yy_PERIPHERAL_MAP), 5-66
peripheral pins, default configuration, 7-7
peripherals, 1-3
and buses, 1-3
compatible with SPI, 13-3
and DMA controller, 5-31
DMA support, 1-4
enabling, 7-2
interrupt generated by, 4-8
interrupts, clearing, 4-13
level-sensitivity of interrupts, 4-15
list of, 1-3
mapping to DMA, 5-103
multiplexing, 7-1
remapping DMA assignment, 5-6
switching from DMA to non-DMA, 5-73
timing, 3-3
used to wake from idle, 4-6
PF0 pin, 7-10
PFx pin, 13-7
phase locked loop, See PLL
pin information, 17-1
pins, 17-1
GPIO, 7-7
multiplexing, 7-1
unused, 17-8
pin terminations, SPORT, 14-8
pipeline, lengths of, 5-51
pipelining
DMA requests, 5-37
PLL, 6-1 to 6-28
active (enabled but bypassed) mode, 6-8
active mode, 6-8
applying power to the PLL, 6-11
block diagram, 6-3
BYPASS bit, 6-9
PLL *(continued)*
CCLK derivation, 6-3
changing clock ratio, 6-6
clock control, 6-1
clock dividers, 6-4
clock multiplier ratios, 6-3
configuration, 6-3
control bits, 6-10
deep sleep mode, 6-9
design overview, 6-2
disabled, 6-11
divide frequency, 6-4
DMA access, 6-8, 6-9
dynamic power management controller (DPMC), 6-7
enabled, 6-11
hibernate state, 6-10
interacting with DPMC, 6-2
and internal clocks, 3-3
maximum performance mode, 6-8
modification in active mode, 6-11
multiplier select (MSEL) field, 6-4
operating modes, operational characteristics, 6-7
operating mode transitions, 6-10, 6-13
PDWN bit, 6-10
PLL_OFF bit, 6-11
PLL status (table), 6-7
power domains, 6-15
power savings by operating mode (table), 6-7
registers, table, 6-19
removing power to the PLL, 6-11
SCLK derivation, 6-2, 6-3
sleep mode, 6-9
STOPCK bit, 6-10
voltage control, 6-7
PLL control (PLL_CTL) register, 6-3, 6-4, 6-19, 6-20

PLL_CTL (PLL control) register, 6-3, 6-4, 6-19, 6-20
PLL divide register, 3-3
PLL_DIV (PLL divide) register, 6-5, 6-19
PLL_LOCKCNT (PLL lock count) register, 6-19, 6-21
PLL_LOCKED bit, 6-20
PLL_OFF bit, 6-20
PLL_STAT (PLL status) register, 6-19, 6-20
PMAP[3:0] field, 5-5, 5-44, 5-66
polarity, GPIO, 7-12
POLC (polarity change) bit, 15-4, 15-25, 15-26
polling DMA registers, 5-51
POLS bit, 15-4, 15-25, 15-26
PORT_CFG[1:0] field, 15-4, 15-26, 15-29
port connection, SPORT, 14-7
PORT_DIR (direction) bit, 15-4, 15-26, 15-29
PORT_EN (enable) bit, 15-26, 15-29
port F
  GPIO, 7-8
  interrupt A channel, 7-16
  peripherals, 7-1
  structure, 7-3
PORTF_FER (function enable) register, 7-6
PORTF_HYSTERESIS register, 7-19
port G
  GPIO, 7-8
  interrupt A channel, 7-16
  peripherals, 7-2, 7-4
  structure, 7-4
PORTG_FER (function enable) register, 7-6
PORTG_HYSTERESIS register, 7-19, 7-20
PORTH_FER (function enable) register, 7-6
port pins, 7-2, 13-38
port pins, test access, B-2
port width, PPI, 15-27
PORTx_FER (function enable) registers, 7-2, 7-3, 7-7, 7-11, 7-22
PORTx_FER registers, 7-22
PORTxIO_BOTH (GPIO set on both edges) registers, 7-26
PORTxIO_BOTH registers, 7-26
PORTxIO_CLEAR (GPIO clear) registers, 7-24
PORTxIO_CLEAR registers, 7-24
PORTxIO_DIR (GPIO direction) registers, 7-22
PORTxIO_DIR registers, 7-22
PORTxIO_EDGE (interrupt sensitivity) registers, 7-26
PORTxIO_EDGE registers, 7-26
PORTxIO (GPIO data) registers, 7-23
PORTxIO_INEN (GPIO input enable) registers, 7-11, 7-23
PORTxIO_INEN registers, 7-23
PORTxIO_MASKA_CLEAR (GPIO mask interrupt A clear) registers, 7-15, 7-30
PORTxIO_MASKA_CLEAR registers, 7-30
PORTxIO_MASKA (GPIO mask interrupt A) registers, 7-27
PORTxIO_MASKA registers, 7-27
PORTxIO_MASKA_SET (GPIO mask interrupt A set) registers, 7-28
PORTxIO_MASKA_SET registers, 7-28
PORTxIO_MASKA_TOGGLE (GPIO mask interrupt A toggle) registers, 7-32
PORTxIO_MASKA_TOGGLE registers, 7-32

PORTxIO_MASKB_CLEAR (GPIO mask interrupt B clear) registers, 7-15, 7-31
PORTxIO_MASKB_CLEAR registers, 7-31
PORTxIO_MASKB (GPIO mask interrupt B) registers, 7-27
PORTxIO_MASKB registers, 7-27
PORTxIO_MASKB_SET (GPIO mask interrupt B set) registers, 7-29
PORTxIO_MASKB_SET registers, 7-29
PORTxIO_MASKB_TOGGLE (GPIO mask interrupt B toggle) registers, 7-33
PORTxIO_MASKB_TOGGLE registers, 7-33
PORTxIO_POLAR (GPIO polarity) registers, 7-25
PORTxIO_POLAR registers, 7-25
PORTxIO registers, 7-23
PORTxIO_SET (GPIO set) registers, 7-24
PORTxIO_SET registers, 7-24
PORTxIO_TOGGLE (GPIO toggle) registers, 7-25
PORTxIO_TOGGLE registers, 7-25
PORTx_MUX (port multiplexer control) register, 7-2, 7-21
PORTx_MUX (port multiplexer control) registers, 7-3, 7-6
PORTx_MUX registers, 7-21
power
  dissipation, 6-15
  domains, 6-15
  plane, 17-5
power management, 1-16, 6-1 to 6-28
power-on reset, 16-3
PPI, 15-2 to 15-37
  active video only mode, 15-10
  block diagram, 15-3
PPI *(continued)*
  clearing DMA completion interrupt, 15-37
  clock input, 15-3
  configure DMA registers, 15-35
  configuring registers, 15-36
  control byte sequences, 15-8
  control signal polarities, 15-25
  data input modes, 15-14 to 15-16
  data movement, 15-9
  data output modes, 15-16 to 15-18
  data width, 15-25
  delay before starting, 15-32
  DMA operation, 15-22
  edge-sensitive inputs, 15-20
  enabling, 15-29, 15-36
  enabling DMA, 15-36
  entire field mode, 15-9
  external frame sync modes, 15-15
  external frame syncs, 15-15, 15-17
  features, 15-2
  FIFO, 15-31
  flow diagram, 15-24
  frame start detect, 15-34
  frame synchronization with ITU-R 656, 15-11
  frame sync polarity with timer peripherals, 15-20
  frame track error, 15-30, 15-33
  general flow for GP modes, 15-14
  general-purpose modes, 15-12
  GP modes, 15-14
  GP modes with frame synchronization, 15-19
  GP output, 15-19
  hardware signalling, 15-16
  horizontal tracking, 15-31
  interlaced video, 15-6
  internal frame sync modes, 15-16
  internal frame syncs, 15-16

PPI *(continued)*
internal frame syncs modes, 15-18
ITU-R 601 recommendation, 15-16
ITU-R 656 modes, 15-5
ITU-R 656 output mode, 15-11
loopback feature, 15-10
memory-mapped registers, 15-25
multiplexed with general-purpose timers, 8-58
no frame syncs modes, 15-14, 15-17
number of lines per frame, 15-33
number of samples, 15-32
operating modes, 15-4, 15-25
overview, 1-9
port width, 15-27
preamble, 15-7
programming model, 15-21
progressive video, 15-6
submodes for ITU-R 656, 15-9
and synchronization with DMA, 15-13
timer pins, 15-20
transfer delay, 15-17
TX modes with external frame syncs, 15-21
TX modes with internal frame syncs, 15-19
valid data detection, 15-15
vertical blanking interval only mode, 15-10
video frame partitioning, 15-7
video processing, 15-5
video streams, 15-8
when data transfer begins, 15-29
PPI_CLK cycle count, 15-32
PPI_CLK pin, 15-3
PPI_CLK signal, 15-25
PPI_CONTROL (PPI control) register, 15-25, 15-26
PPI control register (PPI_CONTROL), 15-25, 15-26
PPI_COUNT[15:0] field, 15-33
PPI_COUNT (transfer count) register, 15-32, 15-33
PPI_DELAY[15:0] field, 15-32
PPI_DELAY (delay count) register, 15-32
PPI_FRAME[15:0] field, 15-34
PPI_FRAME (lines per frame) register, 15-33, 15-34
PPI_FS1 signal, 15-25
PPI_FS2 signal, 15-25
PPI_FS3 signal, 15-31
PPI_STATUS (PPI status) register, 15-29, 15-30
preamble, PPI, 15-7
prescale[6:0] field, 12-25
PRESCALE value, 12-4
priorities
memory DMA operations, 5-46
peripheral DMA operations, 5-46
prioritization
DMA, 5-44 to 5-49
interrupts, 4-4
private instructions, B-4
probes, oscilloscope, 17-6
processor
dynamic power management, 6-1
test features, B-1
processor block diagram, 1-4
program Pxn bit, 7-23
progressive video, 15-6
PS bit, 5-83
PSSE (slave select enable) bit, 13-35, 13-36
public instructions, B-4
public JTAG scan instructions, B-5
PULSE_HI bit, 8-14, 8-16, 8-24, 8-41, 8-45
PULSE_HI toggle mode, 8-15
pulse width count and capture mode, *See* WDTH_CAP mode

pulse width modulation mode, See PWM_OUT mode
pulse width modulation mode, *See* PWM_OUT mode
pulse width modulator, 1-13
PWM_CLK clock, 8-20
PWM_CLK signal, 8-20
PWM_OUT mode, 8-10 to 8-23, 8-43
  control bit and register usage, 8-45
  error prevention, 8-44
  externally clocked, 8-20
  PULSE_HI toggle mode, 8-15
  stopping the timer, 8-21
Pxn bit, 7-22
Pxn both edges bit, 7-26
Pxn direction bit, 7-22
Pxn input enable bit, 7-23
Pxn polarity bits, 7-25
Pxn sensitivity bit, 7-26

Q

quick boot, 16-25

R

RBC bit, 5-37, 5-83
RBSY flag, 13-41
RBSY (receive error) bit, 13-39
RCKFE (clock falling edge select) bit, 14-32, 14-52, 14-55
RCVDATA16[15:0] field, 12-46
RCVDATA8[7:0] field, 12-45
RCVFLUSH (receive buffer flush) bit, 12-36, 12-37
RCVINTLEN (receive buffer interrupt length) bit, 12-36
RCVSERVM (receive FIFO service interrupt mask) bit, 12-40
RCVSERV (receive FIFO service) bit, 12-40, 12-41
RCVSTAT[1:0] field, 12-38
RDTYPE[1:0] field, 14-27, 14-52, 14-54
read/write access bit, 2-5, 2-6
receive buffer[7:0] field, 11-26
receive configuration (SPORTx_RCR1, SPORTx_RCR2) registers, 14-51
receive data[15:0] field, 14-60
receive data[31:16] field, 14-60
receive data buffer[15:0] field, 13-42
receive FIFO, SPORT, 14-58
reception error, SPI, 13-41
register-based DMA, 5-9
registers
  See also registers by name
  system, A-2
REP bit, 5-38, 5-83
request data control command, DMA, 5-34
request data urgent control command, DMA, 5-34
reset
  effect on SPI, 13-15
RESET_DOUBLE bit, 16-52
$\overline{\text{RESET}}$ pin, 16-4
resets
  core and system, 16-71, 16-72
  core double-fault, 16-4
  core-only software, 16-4
  hardware, 16-3, 16-6
  interrupts, 16-6
  power-on, 16-3
  software, 16-5
  system software, 16-3
  watchdog timer, 16-3, 16-5
RESET_SOFTWARE bit, 16-52
reset vector, 16-1
RESET_WDOG bit, 10-5, 16-52
restart control command, DMA, 5-32, 5-33
restart or finish control command, transmit, 5-34, 5-35

restrictions
DMA control commands, 5-34
DMA work unit, 5-24
RETI register, 16-6
RFS pins, 14-30
RFSR (receive frame sync required select) bit, 14-30, 14-31, 14-52, 14-54
RFS signal, 14-18
RFSx signal, 14-5
RLSBIT (receive bit order) bit, 14-52, 14-54
round robin operation, MDMA, 5-47
routing of interrupts, 4-16
ROVF (sticky receive overflow status) bit, 14-59, 14-61
RPOLC (IrDA Rx polarity change) bit, 11-31
RRFST (left/right order) bit, 14-12, 14-53, 14-55
RSCLKx pins, 14-29
RSCLKx signal, 14-5
RSFSE (receive stereo frame sync enable) bit, 14-10, 14-53, 14-55
RSPEN (receive enable) bit, 14-9, 14-51, 14-52, 14-53
RSTART (repeat start) bit, 12-29, 12-30
RUVF (sticky receive underflow status) bit, 14-59, 14-61
RX hold register, 14-59
RX modes with external frame syncs, 15-20
RXNE (receive FIFO not empty status) bit, 14-61
RXREQ signal, 11-7
RXSE (RxSEC enable) bit, 14-53, 14-55
RXS (RX data buffer status) bit, 13-22, 13-39

S

SADDR[6:0] field, 12-28
SAMPLE/PRELOAD instruction, B-6
sampling clock period, UART, 11-8
sampling edge, SPORT, 14-32
SAV codes, 15-31
SAV signal, 15-5
SB (set break) bit, 11-21
scale value[7:0] field, 9-6
scaling, of core timer, 9-7
scan paths, B-5
SCCB bit, 12-25
scheduling, memory DMA, 5-46
SCK signal, 13-4, 13-12, 13-14, 13-15
SCLK, 3-3, 6-5
derivation, 6-2
status by operating mode (table), 6-7
SCLOVR (serial clock override) bit, 12-29, 12-30
SCL pin, 12-5
SCLSEN (serial clock sense) bit, 12-33, 12-34
SCL serial clock, 12-25
SCL (serial clock) signal, 12-4
SCOMPM (slave transfer complete interrupt mask) bit, 12-40
SCOMP (slave transfer complete) bit, 12-40, 12-42
scratch[7:0] field, 11-30
scratchpad memory, and booting, 16-9
SDAOVR (serial data override) bit, 12-29, 12-30
SDA pin, 12-5
SDASEN (serial data sense) bit, 12-33, 12-34
SDA (serial data) signal, 12-4
SDIR (slave transfer direction) bit, 12-28, 12-29
SEN (slave enable) bit, 12-26, 12-27
serial
clock frequency, 13-34
communications, 11-5
data transfer, 14-4

serial *(continued)*
scan paths, B-4
serial clock divide modulus[15:0] field, 14-62
serial peripheral interface, *See* SPI
serial scan paths, B-5
SERRM (slave transfer error interrupt mask) bit, 12-40
SERR (slave transfer error) bit, 12-40, 12-42
set index[5:0] field, 2-5, 2-6
set Pxn bit, 7-24
set Pxn interrupt A enable bit, 7-28
set Pxn interrupt B enable bit, 7-29
shared interrupts, 4-4
SIC, *See* system interrupt controller
SIC_IAR0 (system interrupt assignment 0) register, 4-11
SIC_IMASK (system interrupt mask) register, 4-5
SIC registers, 4-10
signal integrity, 17-3
sine wave input, 1-16
single pulse generation, timer, 8-12
SINITM (slave transfer initiated interrupt mask) bit, 12-40
SINIT (slave transfer initiated) bit, 12-40, 12-42
size of accesses, timer registers, 8-35
SIZE (size of words) bit, 13-35, 13-36
SKIP_EN (skip enable) bit, 15-25, 15-26
SKIP_EO (skip even odd) bit, 15-26, 15-27
slave mode setup, in TWI, 12-11, 12-50
slaves
PAB, 3-5
slave select, SPI, 13-37
slave SPI device, 13-5
sleep mode, 1-17, 6-9
SLEN[4:0] field, 14-48, 14-49, 14-53, 14-54
restrictions, 14-27
word length formula, 14-27
small descriptor mode, DMA, 5-14
small model mode, DMA, 5-68
software
interrupts, 4-3
management of DMA, 5-50
watchdog timer, 1-15, 10-1
software reset, 16-5, 16-52
software reset register (SWRST), 16-52
source channels, memory DMA, 5-7
SOVFM (slave overflow interrupt mask) bit, 12-40
SOVF (slave overflow) bit, 12-40, 12-42
SPE (SPI enable) bit, 13-35, 13-36
SPI, 13-2 to 13-52
beginning and ending transfers, 13-21
block diagram, 13-3, 13-4
clock phase, 13-11, 13-13, 13-16
clock polarity, 13-11, 13-15
clock signal, 13-3, 13-15
code examples, 13-43
compatible peripherals, 13-3
data corruption, avoiding, 13-14
data interrupt, 13-16
data transfer, 13-14
detecting transfer complete, 13-38
and DMA, 13-10
DMA initialization, 13-47
DMA transfers, 13-47
effect of reset, 13-15
enabling the SPI system, 13-35
error interrupt, 13-16
error signals, 13-38 to 13-41
features, 13-2
full-duplex synchronous serial interface, 13-2
general operation, 13-14 to 13-21

SPI *(continued)*
initialization, 13-44
internal interfaces, 13-10
interrupt outputs, 13-16
interrupts, 13-46
master mode, 13-14, 13-17
master mode DMA operation, 13-23
mode fault error, 13-39
multiple slave systems, 13-8
overview, 1-13
reception error, 13-41
registers, table, 13-33
SCK signal, 13-4
slave boot mode, 16-44
slave device, 13-5
slave mode, 13-15, 13-19
slave mode DMA operation, 13-26
slave-select function, 13-37
slave transfer preparation, 13-21
SPI_FLG mapping to port pins, 13-38
starting DMA transfer, 13-49
starting transfer, 13-45
stopping, 13-46
stopping DMA transfers, 13-50
switching between transmit and receive, 13-22
timing, 13-6
transfer formats, 13-11 to 13-13
transfer initiate command, 13-17, 13-18
transfer modes, 13-18
transfer protocol, 13-12, 13-13
transmission error, 13-40
transmission/reception errors, 13-38
transmit collision error, 13-41
using DMA, 13-9
word length, 13-35
SPI_BAUD (SPI baud rate) register, 13-33, 13-34
SPI_BAUD values, 13-34
SPI_CTL (SPI control) register, 13-5, 13-33, 13-35, 13-36
SPI_FLG (SPI flag) register, 13-7, 13-8, 13-33, 13-37
SPIF (SPI finished) bit, 13-9, 13-22, 13-39
SPI_RDBR shadow[15:0] field, 13-43
SPI RDBR shadow (SPI_SHADOW register), 13-33
SPI RDBR shadow (SPI_SHADOW) register, 13-43
SPI_RDBR (SPI receive data buffer) register, 13-33, 13-42
SPI_SHADOW (SPI RDBR shadow) register, 13-33, 13-43
SPI slave select, 13-37
$\overline{\text{SPISS}}$ signal, 13-6, 13-7, 13-8, 13-12
SPI_STAT (SPI status) register, 13-33, 13-38, 13-39
SPI_TDBR (SPI transmit data buffer) register, 13-33, 13-41, 13-42
SPORT, 1-11, 14-1 to 14-73
2X clock recovery control, 14-24
active low vs. active high frame syncs, 14-32
channels, 14-14
clock, 14-29
clock frequency, 14-25, 14-61
clock rate, 14-2
clock rate restrictions, 14-26
companding, 14-28
configuration, 14-10
data formats, 14-27
data word formats, 14-55
disabling, 14-10
DMA data packing, 14-23
enable/disable, 14-9
enabling multichannel mode, 14-17
framed serial transfers, 14-31
framed vs. unframed, 14-30
frame sync, 14-31, 14-34

SPORT *(continued)*
frame sync frequencies, 14-25
framing signals, 14-30
general operation, 14-9
H.100 standard protocol, 14-24
initialization code, 14-54
internal memory access, 14-37
internal vs. external frame syncs, 14-31
late frame sync, 14-17
modes, 14-10
moving data to memory, 14-37
multichannel frame, 14-19
multichannel operation, 14-14 to 14-24
multichannel transfer timing, 14-16
overview, 1-11
packing data, multichannel DMA, 14-23
peripheral access bus error, 14-38
pin/line terminations, 14-8
port connection, 14-7
receive and transmit functions, 14-4
receive clock signal, 14-29
receive FIFO, 14-58
receive word length, 14-59
register writes, 14-45
RX hold register, 14-59
sampling edge, 14-32
selecting bit order, 14-27
serial data communication protocols, 14-2
shortened active pulses, 14-10
signals, 14-5
single clock for both receive and transmit, 14-29
single word transfers, 14-37
stereo serial connection, 14-7
stereo serial frame sync modes, 14-17
stereo serial operation, 14-10
support for standard protocols, 14-24
termination, 14-8
throughput, 14-5

SPORT *(continued)*
timing, 14-38
transmit clock signal, 14-29
transmitter FIFO, 14-56
transmit word length, 14-56
TX hold register, 14-56
TX interrupt, 14-57
unframed data flow, 14-30
unpacking data, multichannel DMA, 14-23
window offset, 14-21
word length, 14-27
SPORT error interrupt, 14-37
SPORT registers, table, 14-44
SPORT RX interrupt, 14-37, 14-59
SPORT TX interrupt, 14-37
SPORTx_CHNL (SPORTx current channel) registers, 14-64
SPORTx_MCMCn (SPORTx multichannel configuration) registers, 14-63
SPORTx_MRCSn (SPORTx multichannel receive select) registers, 14-22, 14-23, 14-65
SPORTx_MTCSn (SPORTx multichannel transmit select) registers, 14-22, 14-23, 14-66
SPORTx_RCLKDIV (SPORTx receive serial clock divider) registers, 14-61
SPORTx_RCR1 (SPORTx receive configuration 1) registers, 14-51
SPORTx_RCR2 (SPORTx receive configuration 2) registers, 14-51, 14-53
SPORTx_RFSDIV (SPORTx receive frame sync divider) registers, 14-62
SPORTx_RX (SPORTx receive data) registers, 14-18, 14-58
SPORTx_STAT (SPORTx status) registers, 14-60

SPORTx_TCLKDIV (SPORTx transmit serial clock divider) registers, 14-61
SPORTx_TCR1 (transmit configuration 1) register, 14-46
SPORTx_TCR2 (transmit configuration 2) register, 14-46
SPORTx_TFSDIV (SPORTx transmit frame sync divider) registers, 14-62
SPORTx_TX (SPORTx transmit data) registers, 14-18, 14-36, 14-56
SSEL[3:0] field, 3-3, 6-5, 6-19
SSEL bit, 17-2
SSEL (system select) bit, 6-19
start address registers
  (DMAx_START_ADDR), 5-74
  (MDMA_yy_START_ADDR), 5-74
STATUS[1:0] field, 11-28, 11-29
STB (stop bits) bit, 11-21
STDVAL (slave transmit data valid) bit, 12-26, 12-27
stereo serial
  data, 14-3
  device, SPORT connection, 14-7
  frame sync modes, 14-17
  operation, SPORT, 14-10
STOPCK (stop clock) bit, 6-20
stop clock (STOPCK) bit, 6-20
STOP (issue stop condition) bit, 12-31
stop mode, DMA, 5-11, 5-68
stopping DMA transfers, 5-28
STP (stick parity) bit, 11-21
streams, memory DMA, 5-7
subbanks
  L1 data memory, 2-3
  L1 instruction memory, 2-2
supervisor mode, 16-6
support, technical or customer, xxxiv
surface-mount capacitors, 17-5
SWRESET bit, 16-54
SWRST, software reset register, 16-52
SWRST (software reset register), 16-52
SYNC bit, 5-24, 5-25, 5-26, 5-61, 5-67, 5-69, 11-17
synchronization
  interrupt-based methods, 5-50
  of descriptor queue, 5-56
  of DMA, 5-50 to 5-60
synchronized transition, DMA, 5-27
synchronous serial data transfer, 14-4
synchronous serial ports, *See* SPORT
SYSCR (system reset configuration register), 16-54, 16-55
SYSCR (system reset configuration) register, 16-54
system
  interrupt controller, 4-2, 8-7
  interrupt processing, 4-8
  interrupts, 4-1, 4-2
  peripherals, 1-3
system and core event mapping (table), 4-3
system clock, 1-16
system clock (SCLK), 6-2
  managing, 17-2
system design, 17-1 to 17-7
  high frequency considerations, 17-3
  recommendations and suggestions, 17-4
  recommended reading, 17-7
system interrupt assignment 0 (SIC_IAR0) register, 4-11
system interrupt controller (SIC), 4-2
  controlling interrupts, 4-4
  enabling flexible interrupt handling, 8-7
  enabling individual peripheral interrupts, 4-4
  main functions of, 4-4
  peripheral interrupt events, 4-17
  registers, 4-10
system interrupt mask (SIC_IMASK) register, 4-5
system peripheral clock, *See* SCLK

system reset, 16-1 to 16-71
SYSTEM_RESET[2:0] field, 16-52
system reset configuration register (SYSCR), 16-54, 16-55
system reset configuration (SYSCR) register, 16-54
system select (SSEL) bit, 6-19
system software reset, 16-3
SZ (send zero) bit, 13-20, 13-36

T

TAP registers
  boundary-scan, B-7
  BYPASS, B-6
  instruction, B-2, B-4
TAP (test access port), B-1, B-2
  controller, B-2
target address, 16-16
TAUTORLD bit, 9-3, 9-5
TCKFE (clock drive/sample edge select) bit, 14-32, 14-47, 14-51
TCNTL (core timer control) register, 9-3, 9-5
TCOUNT (core timer count) register, 9-3, 9-5
TDM interfaces, 14-4
TDTYPE[1:0] field, 14-27, 14-47, 14-49
technical support, xxxiv
TEMT (TSR and UARTx_THR empty) bit, 11-7, 11-24, 11-25
termination, DMA, 5-28
terminations, SPORT pin/line, 14-8
test access port (TAP), B-1, B-2
  controller, B-2
test clock (TCK), B-6
test features, B-1 to B-7
testing circuit boards, B-1, B-6
test-logic-reset state, B-4
test point access, 17-6
TFS pins, 14-30, 14-36
TFSR (transmit frame sync required select) bit, 14-30, 14-31, 14-47, 14-50
TFS signal, 14-18
TFSx signal, 14-5
THRE flag, 11-6, 11-16
THRE (THR empty) bit, 11-11, 11-24, 11-25
throughput
  DMA, 5-41
  from DMA system, 5-40
  general-purpose ports, 7-6
  SPORT, 14-5
TIMDISx bit, 8-36, 8-37
time-division-multiplexed (TDM) mode, 14-14
  See also SPORT, multichannel operation
TIMENx bit, 8-35, 8-36, 14-76
timer configuration (TIMERx_CONFIG) registers, 8-5, 8-40, 8-41
timer counter[15:0] field, 8-42
timer counter[31:16] field, 8-42
timer counter (TIMERx_COUNTER) registers, 8-4, 8-41, 8-42
TIMER_DISABLE bit, 8-45
TIMER_DISABLE (timer disable) register, 8-5, 8-37
timer disable (TIMER_DISABLE) register, 8-5, 8-37
TIMER_ENABLE bit, 8-45
TIMER_ENABLE (timer enable) register, 8-5, 8-35, 8-36, 15-23
timer enable (TIMER_ENABLE) register, 8-5, 8-35, 8-36
timer input select (TIN_SEL) bit, 8-41, 8-46
timer interrupt (TIMILx) bits, 8-4, 8-39
timer period[15:0] field, 8-44
timer period[31:16] field, 8-44
timer period (TIMERx_PERIOD) registers, 8-4, 8-43, 8-44

timers, 8-1 to 8-56
  core, 9-1 to 9-8
  EXT_CLK mode, 8-32
  overview, 1-13
  watchdog, 1-15, 10-1 to 10-10
  WDTH_CAP mode, 11-15
TIMER_STATUS (timer status) register, 8-5, 8-38, 8-39
timer status (TIMER_STATUS) register, 8-5, 8-38, 8-39
timer width[15:0] field, 8-45
timer width[31:16] field, 8-45
timer width (TIMERx_WIDTH) registers, 8-43, 8-45
TIMERx_CONFIG (timer configuration) registers, 8-5, 8-40, 8-41
TIMERx_COUNTER (timer counter) registers, 8-4, 8-41, 8-42
TIMERx_PERIOD (timer period) registers, 8-4, 8-43, 8-44
TIMERx_WIDTH (timer width) registers, 8-43, 8-45
TIMILx (timer interrupt) bits, 8-4, 8-39
timing
  memory DMA, 5-44
  multichannel transfer, 14-16
  peripherals, 3-3
  SPI, 13-6
timing examples, for SPORTs, 14-38
TIMOD[1:0] field, 13-16, 13-18, 13-35, 13-36
TIN_SEL (timer input select) bit, 8-41, 8-46
TINT bit, 9-3, 9-5
TLSBIT (bit order select) bit, 14-47, 14-49
TMODE[1:0] field, 8-11, 8-41, 8-45
TMPWR bit, 9-3, 9-5
TMRCLK input, 8-58
TMREN bit, 9-3, 9-5
TMR pin, 8-46
TMRx pins, 8-3, 8-15
TOGGLE_HI bit, 8-41, 8-46
TOGGLE_HI mode, 8-16
toggle Pxn bit, 7-25
toggle Pxn interrupt A enable bit, 7-32
toggle Pxn interrupt B enable bit, 7-33
tools, development, 1-19
TOVF_ERRx bit, 8-25, 8-28
TOVF_ERRx bits, 8-4, 8-7, 8-15, 8-39, 8-40, 8-47
TOVF (transmit overflow status) bit, 14-57, 14-60, 14-61
TPOLC (IrDA TX polarity change) bit, 11-31
traffic control, DMA, 5-44 to 5-49
transfer count (PPI_COUNT) register, 15-32, 15-33
transfer initiate command, 13-17, 13-18
transfer initiation from SPI master, 13-18
transfer rate
  memory DMA channels, 5-41
  peripheral DMA channels, 5-41
transfers
  memory-to-memory, 5-7
transitions
  continuous DMA, 5-24
  DMA work unit, 5-24
  operating mode, 6-10, 6-12
  synchronized DMA, 5-24
transmission error, SPI, 13-40
transmit clock, serial (TSCLKx) pins, 14-29
transmit collision error, SPI, 13-41
transmit configuration registers (SPORTx_TCR1 and SPORTx_TCR2), 14-46
transmit data[15:0] field, 14-57
transmit data[31:16] field, 14-57
transmit data buffer[15:0] field, 13-42
transmit hold[7:0] field, 11-26

TRFST (left/right order) bit, 14-48, 14-51
triggering DMA transfers, 5-60
TRUNx bits, 8-21, 8-38, 8-39, 8-47
TSCALE (core timer scale) register, 9-3, 9-7
TSCLKx signal, 14-5
TSFSE (transmit stereo frame sync enable) bit, 14-9, 14-10, 14-48, 14-51
TSPEN (transmit enable) bit, 14-46, 14-47, 14-48
TUVF (transmit underflow status) bit, 14-36, 14-57, 14-60, 14-61
TWI, 1-8, 12-2 to 12-56
  block diagram, 12-3
  bus arbitration, 12-8
  clock generation, 12-7
  controller, 12-2
  electrical specifications, 12-56
  fast mode, 12-10
  features, 12-2
  general call address, 12-9
  general setup, 12-10
  I^2C compatibility, 1-8
  master mode clock setup, 12-12
  master mode receive, 12-14
  master mode transmit, 12-12
  peripheral interface, 12-5
  pins, 12-5
  slave mode operation, 12-11
  start and stop conditions, 12-8
  synchronization, 12-7
  transfer protocol, 12-6
TWI_CLKDIV (SCL clock divider) register, 12-25, 12-26
TWI_CONTROL (TWI control) register, 12-4, 12-25
TWI_ENA bit, 12-25
TWI_FIFO_CTL (TWI FIFO control) register, 12-36
TWI_FIFO_STAT (TWI FIFO status) register, 12-38
TWI_INT_STAT (TWI interrupt status) register, 12-40
TWI_MASTER_CTL (TWI master mode control) register, 12-29
TWI_MASTER_STAT (TWI master mode status) register, 12-33
TWI_SLAVE_ADDR (TWI slave mode address) register, 12-28
TWI_SLAVE_CTL (TWI slave mode control) register, 12-26
TWI_SLAVE_STAT (TWI slave mode status) register, 12-28
two-dimensional DMA, 5-11
two-wire interface, *See* TWI
TXCOL flag, 13-41
TXCOL (transmit collision error) bit, 13-39
TXE (transmission error) bit, 13-39, 13-40, 14-57, 14-61
TXF (transmit FIFO full status) bit, 14-60
TX hold register, 14-56
TXHRE (transmit hold register empty) bit, 14-61
TXREQ signal, 11-6
TXSE (TxSEC enable) bit, 14-48, 14-51
TXS (SPI_TDBR data buffer status) bit, 13-22, 13-39

U

UART, 1-14, 11-2 to 11-41
  assigning interrupt priority, 11-12
  autobaud detection, 11-13
  baud rate, 11-7
  baud rate examples, 11-13
  bit rate examples, 11-13
  bit rate generation, 11-12
  bitstream, 11-6
  block diagram, 11-3

UART *(continued)*
booting, 11-14
character transmission, 11-35
clearing interrupt latches, 11-28
clock, 11-12
clock rate, 3-3
code examples, 11-32
connected to peripheral access bus, 11-4
data communication via infrared signals, 11-5
data words, 11-5
divisor reset, 11-30
DMA channels, 11-17
DMA mode, 11-17
errors during reception, 11-7
external interfaces, 11-3
features, 11-2
glitch filtering, 11-9
initialization, 11-32
internal TSR register, 11-6
interrupt conditions, 11-28
interrupts, 11-11
IrDA mode, 11-2
IrDA receiver, 11-9
IrDA receiver pulse detection, 11-10
IrDA transmit pulse, 11-9
IrDA transmitter, 11-8
loopback mode, 11-23
mixing modes, 11-18
modem status, 11-4
non-DMA interrupt operation, 11-38
non-DMA mode, 11-15
polling, 11-37
receive operation, 11-7
receive sampling window, 11-10
registers, table, 11-19
sampling clock period, 11-8
standard, 11-2
string transmission, 11-36

UART *(continued)*
switching from DMA to non-DMA, 11-18
switching from non-DMA to DMA, 11-19
and system DMA, 11-27
transmission, 11-6
transmission SYNC bit use, 11-39
UART ports
overview, 1-14
UART receive buffer registers (UARTx_RBR), 11-7
UARTx_DLH (UART divisor latch high byte) registers, 11-20, 11-30
UARTx_DLL (UART divisor latch low byte) registers, 11-19, 11-30
UARTx_GCTL (UART global control) registers, 11-20, 11-31
UARTx_IER (UART interrupt enable) registers, 11-20, 11-26, 11-27
UARTx_IIR (UART interrupt identification) registers, 11-12, 11-20, 11-29
UARTx_LCR (UART line control) registers, 11-6, 11-20, 11-21
UARTx_LSR (UART line status) registers, 11-20, 11-24
UARTx_MCR (UART modem control) registers, 11-20, 11-23
UARTx_RBR (UART receive buffer) registers, 11-7, 11-19, 11-26
UARTx_SCR (UART scratch) registers, 11-20, 11-30
UARTx_THR (UART transmit holding) registers, 11-6, 11-19, 11-26
UCEN (enable UART clocks) bit, 11-7, 11-12, 11-30, 11-31
UNDR (FIFO underrun) bit, 15-30, 15-31
unframed/framed, serial data, 14-30

universal asynchronous
receiver/transmitter, *See* UART
unused pins, 17-8
urgency threshold enable (UTE) bit, 5-39
user mode, 16-6
UTE (urgency threshold enable) bit, 5-39
UTHE[15:0] field, 5-87

V

VCO, multiplication factors, 6-4
VCO signal, 6-2
VDDEXT pins, 17-4
VDDINT pins, 17-4
VDK, 1-21
vertical blanking, 15-6
vertical blanking interval only submode, 15-10
video frame partitioning, 15-7
video streams
CIF, 15-8
NTSC, 15-5
PAL, 15-5
QCIF, 15-8
VisualDSP++, 16-8
debugger, 1-20
development environment, 1-19
voltage, 6-15
control, 6-7
dynamic control, 6-15
voltage controlled oscillator (VCO), 6-3
voltage regulator control (VR_CTL) register, 6-19, 6-21
VR_CTL (voltage regulator control) register, 6-19, 6-21

W

W1C operations, 5-10
wakeup function, 4-7
watchdog control (WDOG_CTL) register, 10-7, 10-8
watchdog count[15:0] field, 10-6
watchdog count[31:16] field, 10-6
watchdog count (WDOG_CNT) register, 10-5, 10-6
watchdog status[15:0] field, 10-7
watchdog status[31:16] field, 10-7
watchdog status (WDOG_STAT) register, 10-3, 10-4, 10-6, 10-7
watchdog timer, 1-15, 10-1 to 10-10
block diagram, 10-3
disabling, 10-5
and emulation mode, 10-2
enabling with zero value, 10-5
features, 10-2
internal interface, 10-3
overview, 1-15
registers, 10-5
reset, 10-5, 16-3, 16-5
starting, 10-4
waveform generation, pulse width modulation, 8-13
WDEN[7:0] field, 10-7
WDEV[1:0] field, 10-4, 10-7
WDOG_CNT (watchdog count) register, 10-5, 10-6
WDOG_CTL (watchdog control) register, 10-7, 10-8
WDOG_STAT (watchdog status) register, 10-3, 10-4, 10-6, 10-7
WDRESET bit, 16-54
WDSIZE[1:0] field, 5-67, 5-70
WDTH_CAP mode, 8-23, 8-43
control bit and register usage, 8-45
WLS[1:0] field, 11-21
WNR bit, 5-70
WNR (DMA direction) bit, 5-67, 5-70
WOFF[9:0] field, 14-21, 14-64

WOM (write open drain master) bit, 13-14, 13-36
word length
  SPI, 13-35
  SPORT, 14-27
  SPORT receive data, 14-59
  SPORT transmit data, 14-56
work unit
  completion, 5-22
  DMA, 5-14
  interrupt timing, 5-25
  restrictions, 5-24
  transitions, 5-24
write-one-to-clear (W1C) operations, 5-10
write operation, GPIO, 7-9
WSIZE[3:0] field, 14-20, 14-64
WURESET bit, 16-54

X

X_COUNT[15:0] field, 5-75
XFR_TYPE[1:0] field, 15-4, 15-26, 15-29
X_MODIFY[15:0] field, 5-77
XMTDATA16[15:0] field, 12-44
XMTDATA8[7:0] field, 12-43
XMTFLUSH (transmit buffer flush) bit, 12-36, 12-37
XMTINTLEN (transmit buffer interrupt length) bit, 12-36
XMTSERVM (transmit FIFO service interrupt mask) bit, 12-40
XMTSERV (transmit FIFO service) bit, 12-40, 12-41
XMTSTAT[1:0] field, 12-38, 12-39

Y

YCbCr format, 15-27
Y_COUNT[15:0] field, 5-78
Y_MODIFY[15:0] field, 5-79

Z

µ-law companding, 14-23, 14-28